Exploring New Computing Paradigms for Data-Intensive Applications


Doctoral Dissertation
Doctoral Program in Electrical, Electronics and Communications Engineering
(31.st cycle)

Exploring New Computing Paradigms for Data-Intensive Applications

Giulia Santoro

* * * * * *

Supervisor
Prof. Mariagrazia Graziano

Doctoral Examination Committee:
Prof. Alberto Bosio, Referee, Ecole Centrale de Lyon
Prof. Marco Ottavi, Referee, University Of Rome Tor Vergata
Prof. Markus Becherer, TUM
Prof. Massimo Ruo Roch, Politecnico di Torino
Prof. Aida Todri-Sanial, University of Montpellier, CNRS-LIRMM

Politecnico di Torino
June 14, 2019

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Giulia Santoro

Turin, June 14, 2019

Summary

Since the conception of the first stored-program computer by John von Neumann in 1945, processing units have undergone extraordinary transformations. Over the years, computing systems have pervaded numerous aspects of the human life and are now ubiquitous. The continuous evolution of processing systems has been mainly driven by three factors: *technological progress*, *architectural innovation* and an always growing demand for *computational power*. The combined action of these factors has led to the creation of extremely powerful computing systems and to the emergence of some fascinating applications that could not have existed without the support of such computational power. However, such advances do not come for free. *Power consumption* is, nowadays, one of the major concerns. Complexity at the architectural level of processing units, technological scaling and resource demanding characteristics of modern applications all have a large impact on power consumption. Moreover, applications are not only *resource demanding*, but also *data demanding*, and this has a strong effect on the role that the memory plays on power consumption. *Memory accesses* are extremely costly in terms of energy and are a performance bottleneck. In fact, while CMOS technology keeps scaling, memories have not made progress at the same pace. This has created a performance gap between processing units and memories that is best known as *von Neumann bottleneck* or *memory wall*. Memories are not able to provide data at the same rate as processing units are able to compute them. In addition to all these problems, *technological scaling* is also approaching a limit where it would not be possible to further progress because of fundamental physical, technological and economical limitations. The introduction of novel *beyond-CMOS technologies* based on new information-processing paradigms is a potential solution to the limitations of technological scaling. This thesis addresses different aspects of these problems.

In the first part of this research work the need for computational power and energy efficiency is targeted through a specific and widespread application: Convolutional Neural Networks (CNNs). Being resource and data demanding, CNNs require energy efficient hardware acceleration. For this aim, a custom-designed hardware accelerator is proposed. The *Deep Learning Processor* combines the quality of design achieved with the ASIC implementation flow with the reconfigurability of FPGAs. The accelerator is an array of Processing Elements interconnected through

a Network-on-Chip. The Processing Element is the basic block of the whole accelerator and it has been designed to be *flexible* but at the same time *optimized for CNN-like workload*, *performance oriented* and *low power*. The Deep Learning Processor has been used as an architectural template for conducting a design space exploration that takes into account all the key features of the accelerator and defines the best configurations in terms of energy efficiency and throughput.

In the second part of this thesis, the limitations related to the technological scaling and the von Neumann bottleneck are targeted through the exploration of a *non-von-Neumann computing paradigm* that is *Logic-in-Memory*. This novel approach goes beyond the separation between computation and memory, typical of von Neumann processing systems, trying to fully integrate them in a single unit. Data are computed directly inside the memory without the need to move them. This approach has a twofold advantage: tearing down memory accesses (and the related power consumption) and demolishing the memory wall. This research work investigates the concept of Logic-in-Memory by presenting a novel *Configurable Logic-in-Memory Architecture* (CLiMA) that exploits the in-memory computing paradigm while also targeting flexibility and high performance. A version of CLiMA based on an emerging non-CMOS technology, namely Nano Magnetic Logic, is also presented. The effectiveness of the CLiMA approach is validated through comparisons with the non-LiM architecture presented in the first part of this thesis. Moreover, a *taxonomy* that classifies the main works found in literature regarding the in-memory processing topic is presented.

We can only see a short distance ahead, but we can see plenty there that needs to be done.

A. M. TURING
Computer Machinery and Intelligence, 1950

Contents

Part I

Hardware Acceleration for Deep Neural Networks

Chapter 1

Motivations and background

The idea of a learning machine dates back to 1947, when Alan Turing, during a lecture given to the London Mathematical Society, predicts the existence of a "machine that can learn from experience" [114]. Few years later, in the well-known paper *Computer Machinery and Intelligence* [113], Turing tries to answer the question "Can machines think?" introducing the famous Imitation Game. He does not give an exact answer to weather machines might show intelligence or not, but he defines aspects and characteristics of a hypothetical learning machine, foretelling a time where machines will compete with human beings in any task.

Since then, Artificial Intelligence (AI) has undergone drastic advances in all fields thanks to new computing technologies. Among all AI fields, Machine Learning is one of the most studied and used to solve disparate class of problems.

Machine Learning comprises a wide family of algorithms: in particular, this work focuses on Deep Learning for image recognition and classification.
Deep Learning is at the base of Convolutional Neural Networks, learning models that are being successfully used for this purpose as they are highly accurate in classifying images. However, Convolutional Neural Networks are resource, power and time demanding, hence, they require powerful and efficient systems to accelerate the computation. This work explores circuit techniques and design methodologies for energy efficient Deep Learning computation by taking as reference a custom-designed hardware accelerator.

1.1 Machine Learning

Machine Learning (ML) is the field of AI that deals with the development of learning techniques [96] that makes it possible for a computer to learn and improve at fulfilling a task.
ML theory is vast and, since it is not the focus of this work, it will not be treated from a mathematical point of view but only some basic concepts will be given to

understand what will be described in the next chapters of this thesis.

1.1.1 Learning Methods

ML is based on different learning methods. The main ones are:

- Supervised learning [94][75]: its task is to learn a function that maps an input to an output based on example pairs that are given as input. A pair is made of an input and a label, i.e. the desired output. The example input pairs, referred to as labeled training data set, are fed to the learning algorithm. By analyzing the training data set, it infers a function that can be used to map new input data. If the learning method works correctly it should assign to new unseen examples the expected output labels. Classification problems, for example, make use of supervised learning.

- Unsupervised learning [95][75]: unlike the supervised case, example inputs are not labeled. Here the objective is to deduce an intrinsic structure or some common attributes from input data. Clustering, as instance, belongs to this class.

- Reinforcement learning [108][75]: this method is based on trial and error. Differently from supervised learning it does not have labeled training data and unlike unsupervised learning it is characterized by a reward measure which has to be maximized during the trial and error steps. Decision-making problems make use of this kind of learning method.

In the rest of this work only supervised learning will be considered.

1.1.2 Working principle

The key point in ML is that there are patterns underlying the data that can be used to construct an approximation of the process that generated this data [4]. ML techniques are based on representation learning [12], a method that aims at learning data representations with the goal of ease the extraction of useful information when building prediction models. In order to understand how ML algorithms work, it could be useful to use a similarity with part of the learning process of a child. In its first years of life, a child needs to see things in order to learn what they are and associate a name to them. Once he gains this knowledge, he will be able to recognize objects, animals, people and so forth. In case of behavior-related knowledge, the child will need experience to learn. The working principle of a ML algorithm, shown in figure 1.1, is basically the same: it learns a model from example data or past experience (the training set) and it optimizes it, that is the training phase. After that, the model built can be inferred to unseen data or problems (new input), that

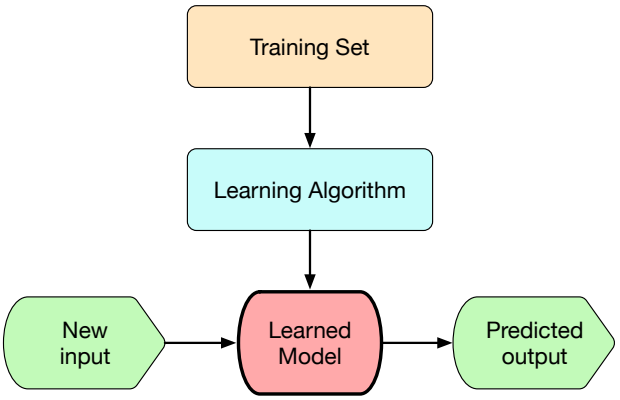

Figure 1.1: Machine Learning working principle scheme.

is the inference phase. If the training phase was robust and the learned model accurate, the predicted output should be faithful to the real expected output.

1.1.3 Applications

ML algorithms are nowadays ubiquitous. Possible applications cover a wide range of fields: from computer vision to medical diagnosis [68] and health-care, from general game playing [104] to robot motion [8], from bioinformatics to natural language processing [100] and so on.
Some practical examples of ML applications in real fields follow.

- Face/object recognition: this computer vision technique allows to recognize faces and objects in images or videos. Detection of faces or objects comes at hand in fields such as security surveillance or manufacturing quality control, respectively. It could also be applied in medical imaging for the detection of tumors, as instance.

- Customer segmentation: in the marketing field, the way to target specific groups of clients is customer segmentation. Clients are clustered into different groups based on the identification of some common characteristics such as age, gender, interests, income level and so on. Each of these groups are addressed with a different marketing strategy in order to improve the quantity of purchases or customer loyalty and satisfaction.

- Autonomous driving: this refers to a vehicle that is able to move autonomously. Being aware of what happens (semaphores, pedestrians, other cars) while driving a car and taking decisions on the fly based on the ever changing environment is arduous. For this reason and for their safety critical nature, self-driving vehicles are still a challenge but ML algorithms have paved the way for significant improvements.

These are just few examples of the numerous potential applications in which ML techniques can be successfully applied.

1.1.4 Deep Learning

Machine Learning comprises a wide family of algorithms. Depending on the application, the type of dataset and the accuracy required for the learned model, one might choose a specific kind of algorithm rather than another.
Among the learning models that belong to ML, Deep Learning (DL) is one of the most promising: in the last years, indeed, the advances done in this fields have revolutionized the world of AI. For sake of clarity, figure 1.2 depicts the relation between AI, ML and DL.

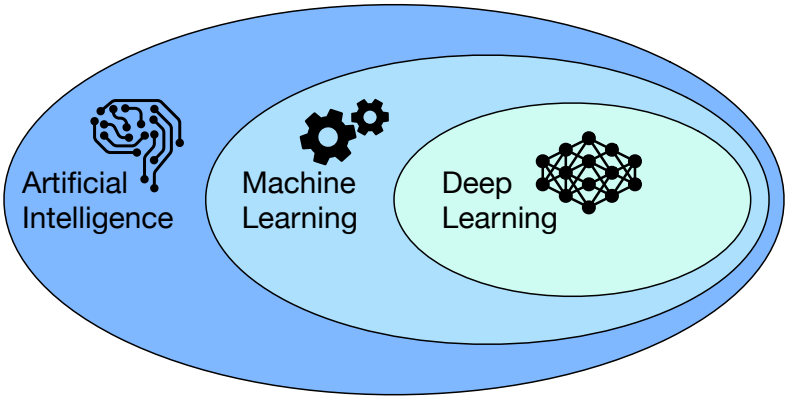

Figure 1.2: Relation between AI, ML and DL.

The DL computational model is based on the automatic learning of data features and representation and it is, nowadays, widely and successfully used in disparate fields such as image recognition [69], natural language processing [25], bioinformatics, strategic marketing and many more.
The key concept behind DL is that it learns multiple levels or a hierarchy of data features [73], that is the reason of being defined deep. Figure 1.3 depicts the feature extraction approach adopted in DL models: starting from raw input data, several features with different abstraction levels are extracted and learned. All of them, from low level (none or 'shallow' abstraction) to high level ('deep' abstraction) ones, contribute to the definition of very complex learning functions.
ML and DL differ in many aspects:

1. Feature extraction: in ML algorithms features are extracted and selected manually while in DL they are identified and learned automatically.

2. Data dependency: DL techniques, differently from ML ones, require a huge quantity of data to perform at their best. Hence, DL scales well with data-intensive applications.

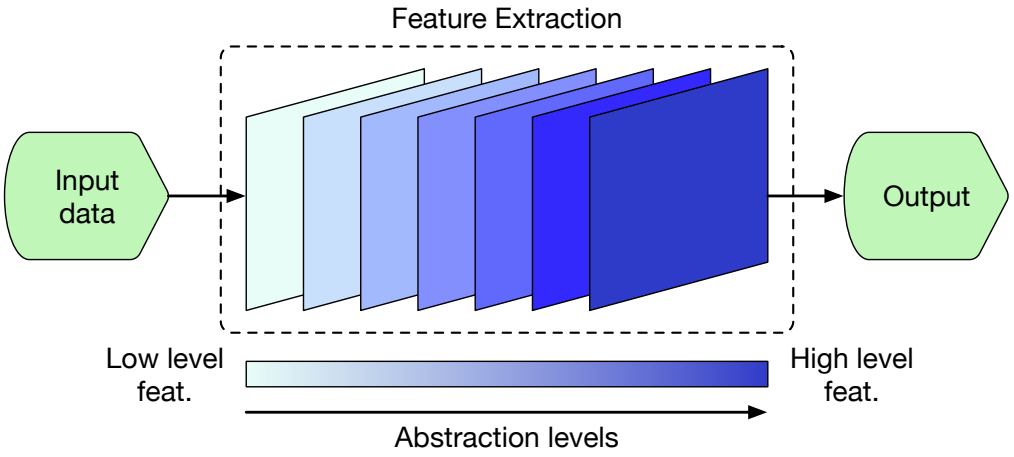

Figure 1.3: Feature extraction in DL models involves multiple levels of data representation abstraction.

3. Computational power: given the large data quantity needed, DL models require high computational power and high-end machines to run (e.g. GPUs) while ML algorithms can work on low-end machines.

4. Training time: in general, the training phase of DL algorithms, unlike in ML, is highly time demanding because a large amount of parameters must be learned.

5. Results interpretation: DL models are very complex and, in turn, it is not possible to fully understand why they behave in a certain way. The same cannot be said about ML models that are based on rules that help understanding the reasoning behind their choices.

These differences suggest that the choice of using DL rather than ML depends on the kind of problem to face, on the quantity and type of input data and on the computational power available.

1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a subset of Machine Learning models. ANNs can be shallow or deep, in this case they are referred to as Deep ANNs (or simply Deep Neural Networks, DNNs). ANNs are vaguely inspired to the human brain neural network.
These kind of models are used when the problem to deal with involves a large number of features or when the input dataset is complex. ANNs are heavily used nowadays in the most disparate fields.

1.2.1 The single-layer perceptron

In the human brain there are neurons and synapses: the former can be seen as processing units working in parallel, the latter are the interconnections among these units. ANNs have a similar structure.
In particular, the basic processing element of an ANN is the perceptron [92], a unit that behaves analogously to the human brain perception process. A perceptron, depicted in figure 1.4.A, has inputs ($x_i,\ i = 0, \ldots, n$) that can either come from the outside environment or that can be the outputs of other perceptrons. Each input

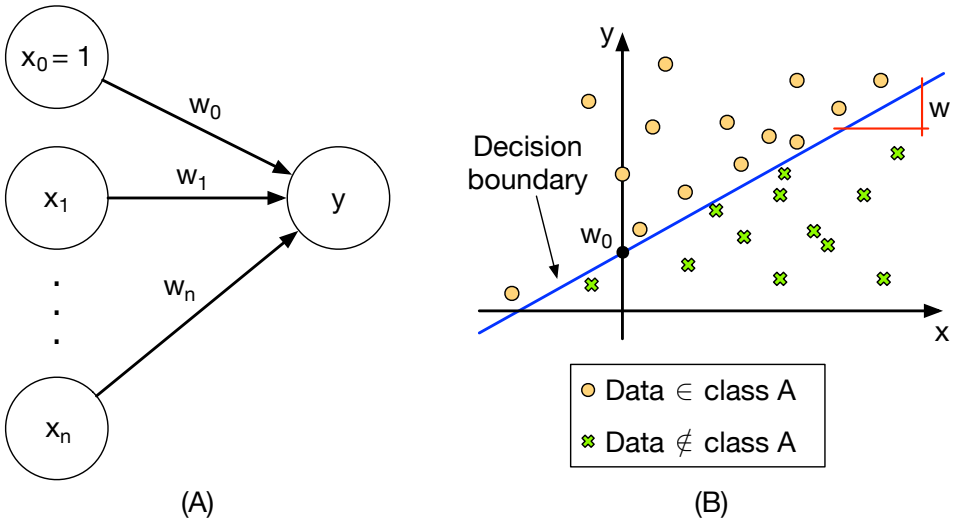

Figure 1.4: (A) Perceptron representation. $x_i,\ i = 1, \ldots, n$ are inputs while y is the perceptron output. $w_i,\ i = 1, \ldots, n$ are the synaptic weights of the connections between inputs and output. (B) Linear discrimination.

is connected to the output (y) by means of a synaptic weight ($w_i,\ i = 0, \ldots, n$). w_0 is called bias unit and it is always equal to 1. For this reason, usually it is omitted. This simple perceptron has only one layer of synaptic weights, that is why it is also called single-layer perceptron. In the basic case, the output of the perceptron is equal to the weighted sum of the outputs [15]:

$$y(\mathbf{x}) = \sum_{i=1}^{n} x_i w_i + w_0 \tag{1.1}$$

The function that links x and y is linear and equation 1.1 is a line with slope w and intercept w_0. As shown in picture 1.4.B, this simple function works as a decision boundary and it can be used as a binary classifier since it divides the xy plane into two. The simple perceptron can be used to separate data that belongs to group A,

from data that do not. As a result, $y(\mathbf{x})$ can be defined as a threshold function:

$$\text{data} \begin{cases} \in \text{ class A} & \text{when } y(\mathbf{x}) > 0 \\ \notin \text{ class A} & \text{otherwise} \end{cases}$$

When considering n dimensions, y becomes a dot product between vectors:

$$y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \tag{1.2}$$

with $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$. $\mathbf{w}$ are the weights to learn. In general, y is called *activation function* or *perceptron hypothesis*.
The perceptron depicted in figure 1.4.A can only perform a binary discrimination: either an input belongs to a certain class or not. When dealing with multi-class classification problems, more perceptrons working in parallel can be used [15] (figure 1.5). In this case, each perceptron $y_i,\ i = 0, \ldots, k$, is associated with a function

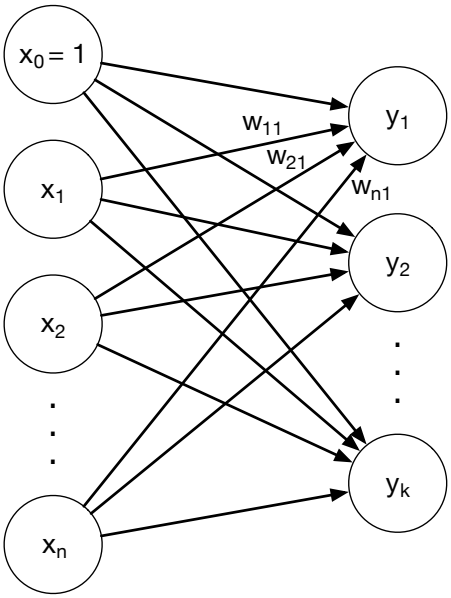

Figure 1.5: Multi-class perceptron.

$f(\mathbf{x})$ with weight vector $\mathbf{w}_i,\ i = 0, \ldots, k$. Equation 1.2 can be rewritten as:

$$y_i(\mathbf{x}) = \sum_{j=1}^{n} x_j w_{ij} + w_{i0} = \mathbf{w}_i^T \mathbf{x},\ i = 1, \ldots, k \tag{1.3}$$

w_{ij} is the synaptic weight that connects input x_j to output y_i. All the synaptic weight vectors constitute the weight matrix $\mathbf{W}$.

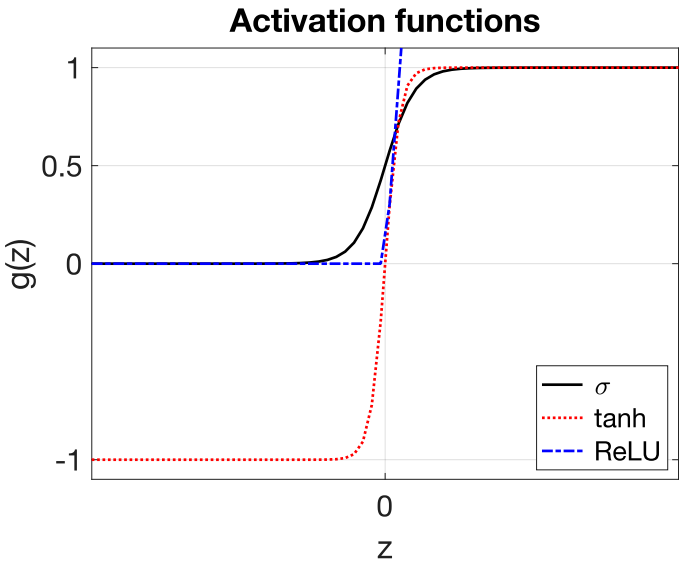

Figure 1.6: Most used activation functions: logistic sigmoid σ, hyperbolic tangent tanh and ReLU.

1.2.2 Activation Functions

When classes are not linearly separable, other kinds of functions must be defined. Among the most used (shown in graph 1.6) there are:

- Logistic sigmoid [47] $\sigma(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent [47] $f(z) = \tanh(z)$
- Rectified Linear Unit (ReLU) [27][39] $f(z) = max(0, z)$

If $z = \mathbf{w}^T\mathbf{x}$ then the activation y can be expressed as:

- a *sigmoid* activation

$$y(z) = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}} \tag{1.4}$$

- a *hyperbolic tangent* activation

$$y(z) = \tanh(\mathbf{w}^T\mathbf{x}) \tag{1.5}$$

- a *ReLU* activation

$$y(z) = max(0, \mathbf{w}^T\mathbf{x}) \tag{1.6}$$

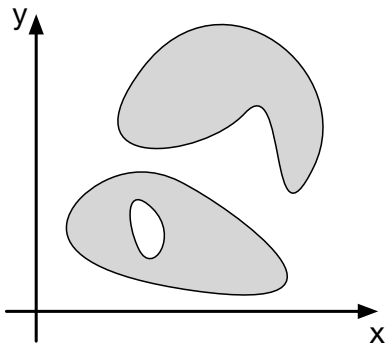

Figure 1.7: Complex decision boundary.

1.2.3 The multi-layer perceptron

When the aim is to build more complex functions to fit complicated hypothesis like the one shown in picture 1.7, multi-layer perceptrons (MLPs) must be used [15]. These kind of perceptrons, as shown in picture 1.8, are made of at least two layers of synaptic weights. The units that connect inputs to outputs through intermediate layers of synaptic weights are called *hidden units*. $a_j^{(k)},\ j = 1, \ldots, m,\ k = 1, \ldots, d$

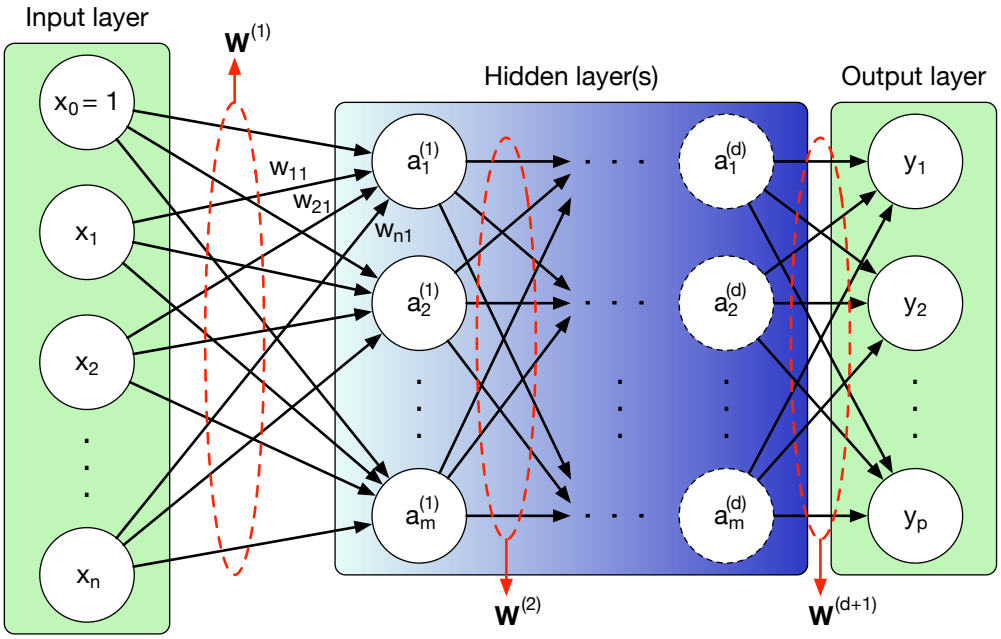

Figure 1.8: Multi-layer perceptron.

is the *activation unit* (or simply *activation*) associated with the j-th hidden unit in the k-th layer. $\mathbf{W}^{(k)}$ is the weight matrix that maps the outputs of a certain layer with the inputs of a subsequent one. A MLP with two or more layers of perceptrons

is an ANN. If an ANN has a large number of hidden layers than it is a deep neural network.
First layer perceptrons satisfy the following equation:

$$a_j^{(1)} = g(\mathbf{w}_j^{(1)T}\mathbf{x}),\ j = 1, \ldots, m \tag{1.7}$$

where $g(\cdot)$ is a generic activation function.
Equation 1.7 can be rewritten as:

$$\mathbf{a}^{(1)} = g(\mathbf{W}^{(1)}\mathbf{x}) \tag{1.8}$$

Hidden perceptrons, instead, satisfy the following equation:

$$a_j^{(k)} = g(\mathbf{w}_j^{(k)T}\mathbf{a}_j^{(k-1)}),\ j = 1, \ldots, m,\ k = 2, \ldots, d \tag{1.9}$$

Finally, output layer activations can be calculated as:

$$y_z = g(\mathbf{w}_z^{(d+1)T}\mathbf{a}_j^{(d)}),\ j = 1, \ldots, m,\ z = 1, \ldots, p \tag{1.10}$$

It can be observed, by substituting equations 1.7, 1.9 and 1.10 one into the other, that output activations are calculated by taking into account all intermediate and input activations. This method is called *forward propagation* since input values are forwarded through intermediate layers to the output layer.
Using vector notation, equation 1.10 can be rewritten as:

$$\mathbf{y} = g(\mathbf{w}^{(d+1)T}\mathbf{a}_j^{(d)}),\ j = 1, \ldots, m \tag{1.11}$$

Equation 1.11 represents the ANN *hypothesis*. The weights are called network *parameters* and they are learned by the ANN during the training phase. Learning these parameters allows to build an accurate hypothesis that can suitably fit the type of problem taken into account. The deeper the network – i.e. more hidden layers – the more accurate the hypothesis but the larger the number of parameters and the overall complexity of the ANN.

1.2.4 Brief Overview on Learning

There exists various learning algorithms that are used to train ANNs. However, since they are not the focus of this work, they will not be thoroughly treated in this contest. Just few notions will be mentioned so as to give an idea to the reader of what learning is.
The working principle of learning algorithms is explained in figure 1.9. Learning is based on the minimization of a *cost function* that is defined as the difference between the actual output and the predicted output (i.e. the output computed by the ANN). The cost function is, basically, the error that the network does when

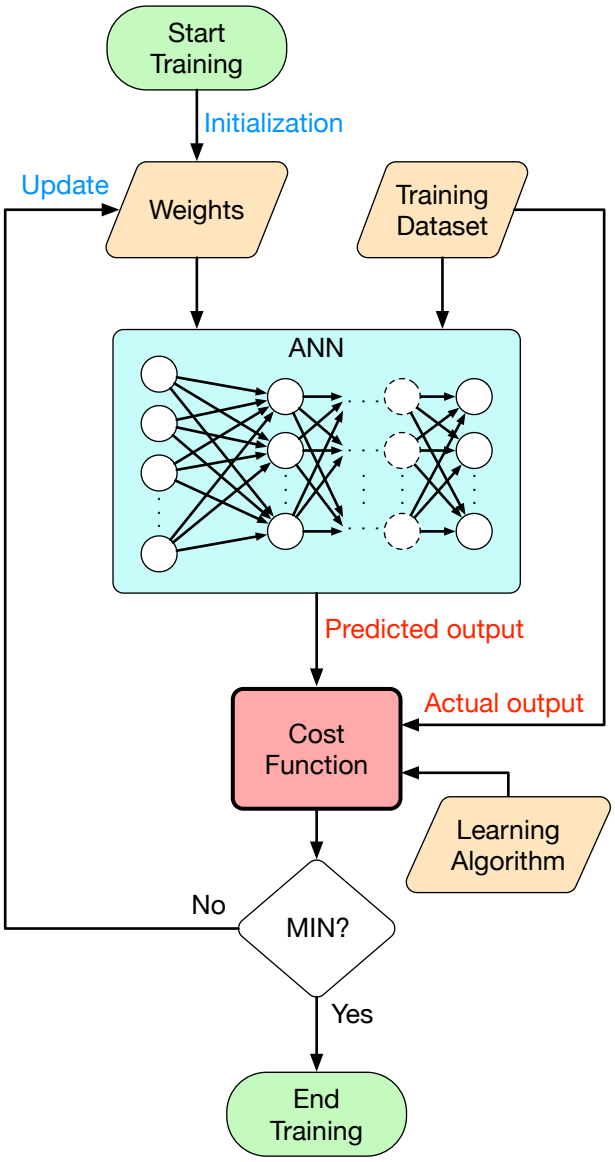

Figure 1.9: Learning iteration principle in ANNs.

carrying out the classification. The predicted output is obtained by feeding the ANN with the training dataset, while the actual output, considering a supervised learning model and a given dataset, is known *a priori*. This minimization phase is an iterative procedure that, step after step, updates the network weights until the cost function has reached a minimum value. Weights are updated on the basis of the error that is propagated backward through all the network layers. At the end, the learning stage returns a trained ANN whose parameters (the weights) are the ones that the network has learned and that better map inputs to the desired output.

1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of DNNs that are particularly suitable for pattern recognition and classification. In fact, in the last ten years, they have been successfully deployed for complex tasks such as image recognition and classification, video analysis, natural language processing, sound perception and more. Performing this kind of tasks require a system that is able to extract relevant representations of the input (e.g. an image) while remaining insensible to variations and distortions [74]. CNNs are specifically designed to automatically learn invariant representations (also called *features*) by using discrete convolution (this is the reason why these networks are called convolutional). In the contest of this thesis the focus is on CNNs applied to image detection.

The working principle of CNNs is neurobiologically inspired by the pioneering work led by neuroscientists Hubel and Wiesel on the visual cortex of a cat [54]. Cortical neurons are locally responsive, meaning that they react to stimuli only in a delimited region of the visual field, called *receptive field*. In order to cover the whole visual field, receptive fields of different neurons partially overlap. Artificial neurons in CNNs (which equivalent to perceptrons) act in the same way.

1.3.1 Main Characteristics

As shown in figure 1.10, layers in a Convolutional Neural Network are three-dimensional, in fact, they are made of several 2D neuron planes. CNNs peculiar characteristics, that distinguish them from other kinds of neural networks, are mainly three [72].

1. *Receptive fields locality*: each neuron in a layer of a CNN is connected only to a small portion of the input image, exploiting the spatial locality propriety of neurons in the brain visual cortex. This means that each neuron extracts some key features (as instance, edges or corners) only from a small region of the whole image.

2. *Weight sharing*: features detected by a neuron in its receptive field are likely to be found in the whole image. For this reason, all other neurons, whose receptive fields focus on different regions, share the same weights that are used to filter the input image and extract crucial features when performing recognition.

3. *Subsampling*: the idea behind subsampling is that the exact location of a feature, once discriminated, becomes less important than its relative position with respect to the rest of the features. This is why subsampling can be exploited to reduce the resolution of the filtered image.

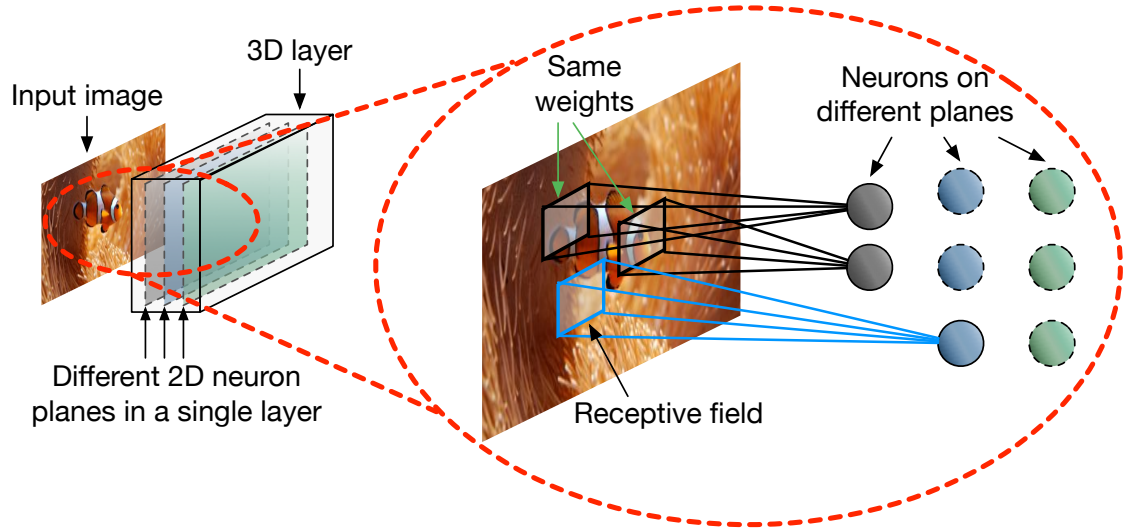

Figure 1.10: CNN layers are 3D: they contain several 2D planes of neurons. Each neuron in a CNN extract features from a defined region of the input image using learned weights. This region is called receptive field. Neurons in the same plane share weights.

1.3.2 Generic Architecture

A typical CNN architecture, as depicted in figure 1.11, is a sequence of 3D layers, where each layer can be seen as a volume of neurons. Each plane of neurons

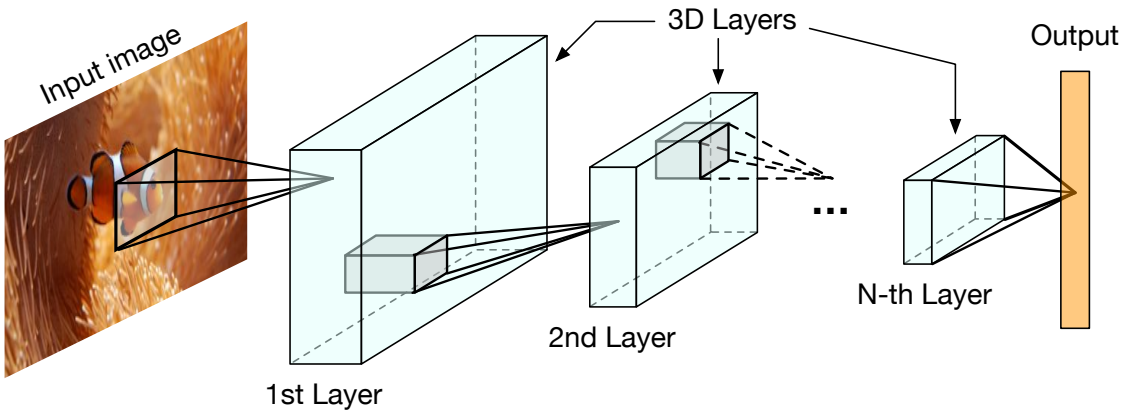

Figure 1.11: Typical architecture of a CNN. Multiple 3D layers of neurons are used to filter and extract key features from the input image.

inside a layer performs a computation that generates a 2D output called *feature map* [70], as a result, the output of a single layer is a volume of feature maps. These maps carry information related to the features detected by neurons when

analyzing the image. Output feature maps of a certain layer become the input of the subsequent layer. This sequence of feature maps elaboration, layer after layer, allows to combine lower-level features into higher-level features [73], following a bottom-up approach. Indeed, an image can be decomposed in different features: groups of edges and corners (low-level features) assemble into patterns, patterns (mid-level features) constitutes parts of an object and parts (high-level features) form the object itself. The last layers of a CNN (more details will be given in the next subsection) perform the actual classification. The output of a CNN is a probability distribution over the classes that the network is able to discriminate.

Main Layers

Layers in a typical CNN can be distinguished in three main kinds [74], namely *convolution*, *rectification* and *pooling*. Figure 1.12 shows the main operations in a CNN. The input image is represented as a matrix of pixels (unsigned integers that can assume values in the range $[0, 255]$). A more detailed description of the computation performed in each layer will be given in the following.

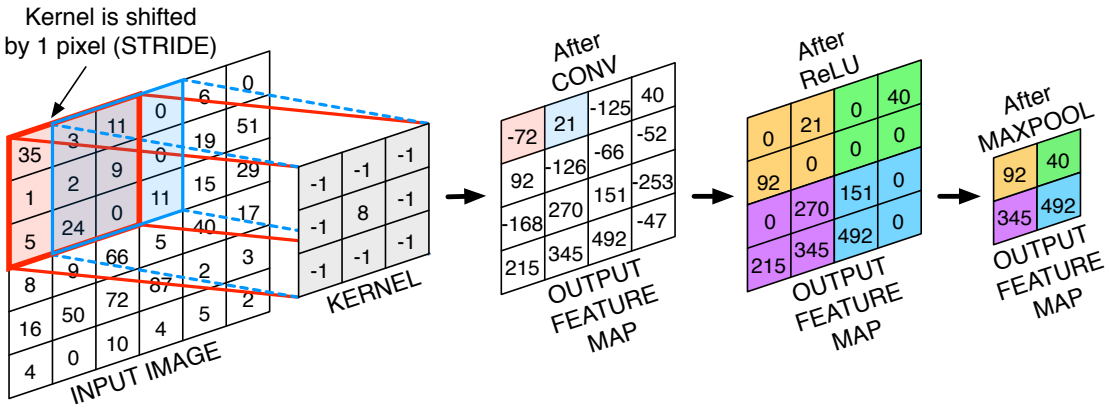

Figure 1.12: Main operations in a CNN.

Convolution A convolution layer (CONV), as the name suggests, is where neurons perform convolutions between the input image and the learned weights. Each neuron is associated with a *kernel* (i.e. a matrix of weights). The kernel is shifted all over the input image, computing a weighted sum of the inputs. Observing the example in figure 1.12, the kernel is a 3×3 matrix that is first applied on the red region of the input image, also called *convolution window*, performing an element-wise multiplication and a final accumulation of all the obtained values. If I is the input image and K is the kernel, then the convolution operation can be expressed

as:

$$o = \sum_{i=1}^{3}\sum_{j=1}^{3} I[i][j] \cdot K[i][j] \tag{1.12}$$

The output value o is the first element (in picture 1.12 it is the red element in the output feature map after the convolution) of the output feature map produced. Then the kernel is shifted by a variable quantity called *stride* (in figure 1.12 is equal to 1) covering a new convolution window and a new output is produced. The whole image is scanned following this procedure and the result is an output feature map that becomes the input for the next layer. The process of shifting the kernel on the input image is usually referred to as *sliding window* process. The size of the output feature map depends on the value of the stride. This parameter is used to downsample the input image [70] loosing some position information, as already explained in subsection 1.3.1.
Figure 1.13 generalizes the operation of convolution between an image of size R×C and a kernel of size K×K with stride S. The output feature map resulting from the

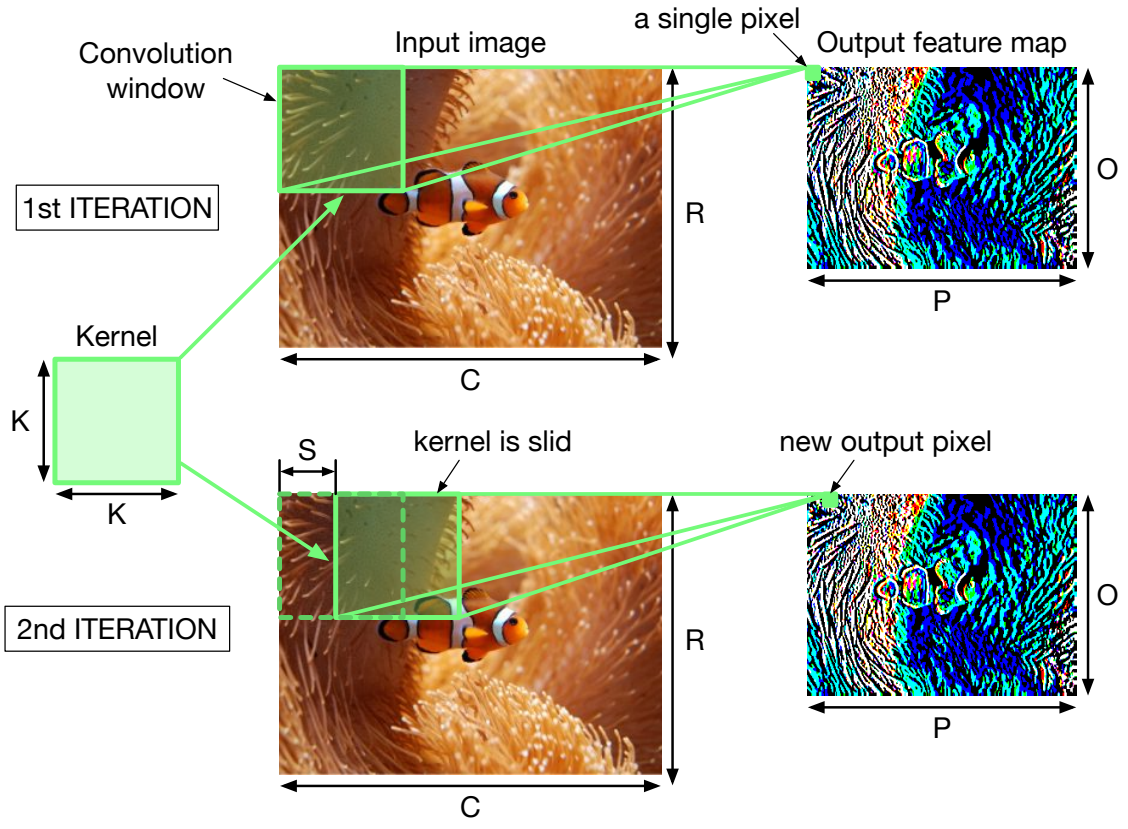

Figure 1.13: Basic convolution operation. The kernel is slid until all the input image has been filtered. The result of the convolution is an output feature map that highlights certain features depending on the type of filter used.

convolution has dimensions O×P that can be calculated as:

$$O = \frac{M - k1}{S} + 1; \; P = \frac{N - k2}{S} + 1 \tag{1.13}$$

Usually kernel, input image and output feature map are square shaped. The example in figure 1.13 is a simplification of what happens in real CNNs where the convolution is *high-dimensional*. Figure 1.14 depicts a more general case. First of

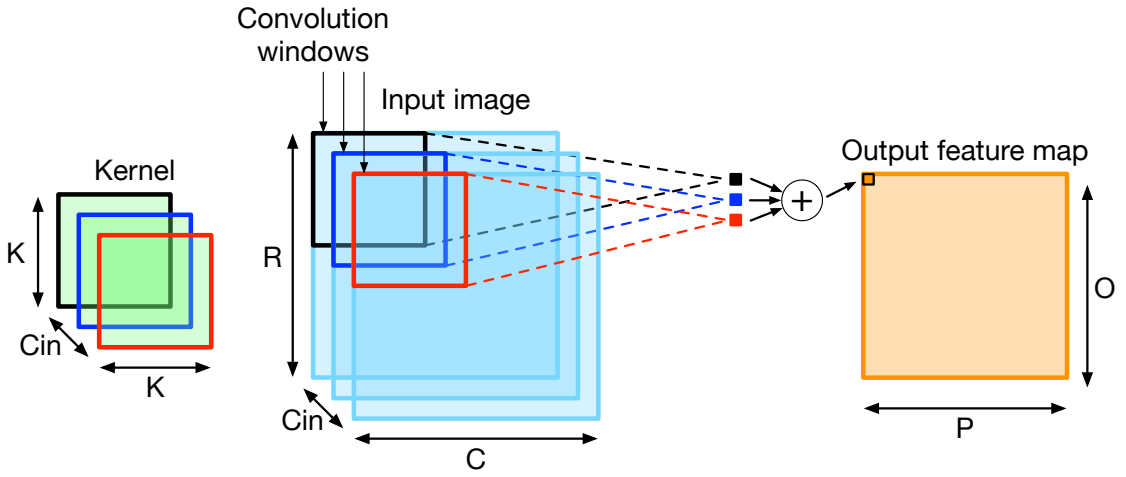

Figure 1.14: High-dimensional convolution with a single kernel. Each convolution window produces an output value. The values are summed together in order to compute the final pixel of the output feature map.

all, the input image is RGB, hence, it is composed of three channels (red, green and blue). The parameter C_{in} represents the number of *input channels*. The kernel, also, is a 3D matrix with C_{in} channels as the input image. Each kernel channel is slid on the correspondent input image channel. For each convolution window, the output values obtained by convolving each kernel channel with the correspondent image channel are summed together to produce the final pixel of the output feature map. Moreover, as said in subsection 1.3.2, layers are made of many neuron planes each with its own kernel matrix. So each layer has multiple kernels to detect multiple features from the same image as shown in picture 1.15, where the number of different kernels is indicated by F. For each kernel, a channel of the output feature map is produced. The parameter C_{out} indicates the number of output channels from a layer that is equal to the number of convolution kernels (F in figure 1.15). Since the output feature map of a layer becomes the input feature map for the next layer, the high-dimensional convolution operation described since now is generally valid for any feature map (not only for the input image). A complete convolution between a R×C×C_{in} feature map and F kernels of size K×K×C_{in} with stride S is defined by the following pseudo-code:

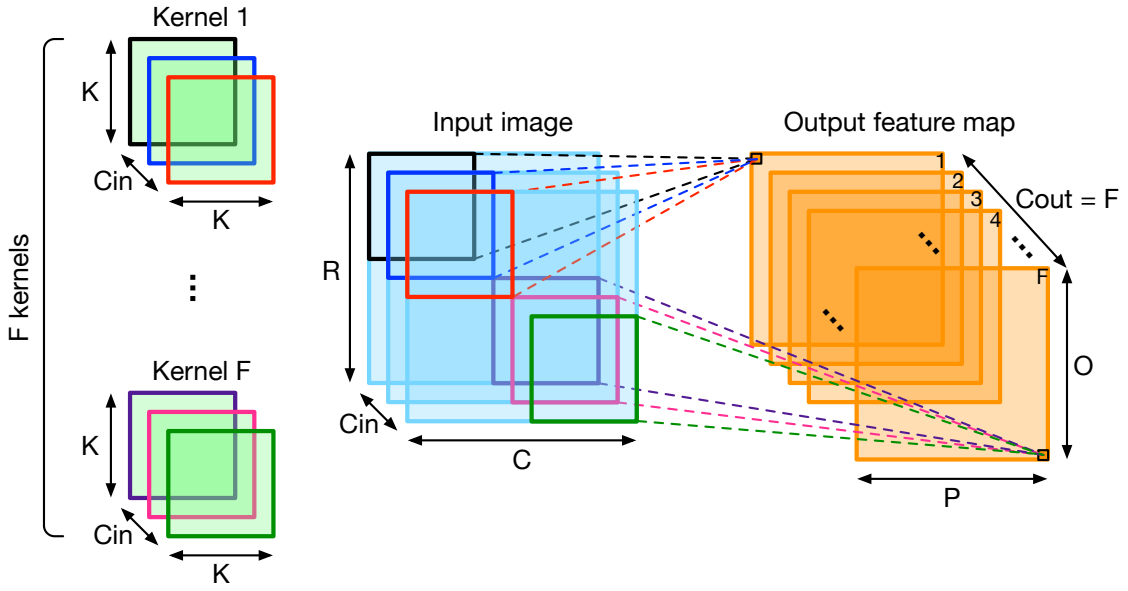

Figure 1.15: High-dimensional convolution with multiple kernels. The output feature map has multiple channels (C_{out}) as the number of kernels (F).

```
for (row=0; row<O; row++){
 for (col=0; col<P; col++){
  for (m=0; m<C_out; m++){
   for (n=0; n<C_in; n++){
    for (i=0; i<K; i++){
     for (j=0; j<K; j++){
      out_fm[row][col][m]+=kernel[i][j][n][m]×
       in_fm[S·row+i][S·col+j][n];
}}}}}}
```

Listing 1.1: High-dimensional convolution pseudo code.

Rectification As said in subsection 1.2.2, there exists many activation functions. Among those, one of the most used is the Rectifying Linear Unit (ReLU) because it accelerates training of Deep CNNs of several times with respect to other activation functions [39][69]. It basically saturates to 0 every negative value in the output feature map as shown in figure 1.12.

Pooling Feature pooling is a form of down-sampling and it is used in CNNs because it introduces invariance to image variations such as translations, deformations, noise [16][42]. One of the most used form of feature pooling is *max pooling* (MAXPOOL) since it works better than other forms of pooling [69]. As depicted

in figure 1.12, it divides the feature map in regions (pools) from which it select the maximum value producing a more compact output feature map. The aim of pooling, indeed, is also to reduce the size of feature maps in order to lighten the computational load of the network.

Figure 1.16 shows a complete CNN architecture. The layers described above are repeated different times inside a deep convolutional network. It is also important to

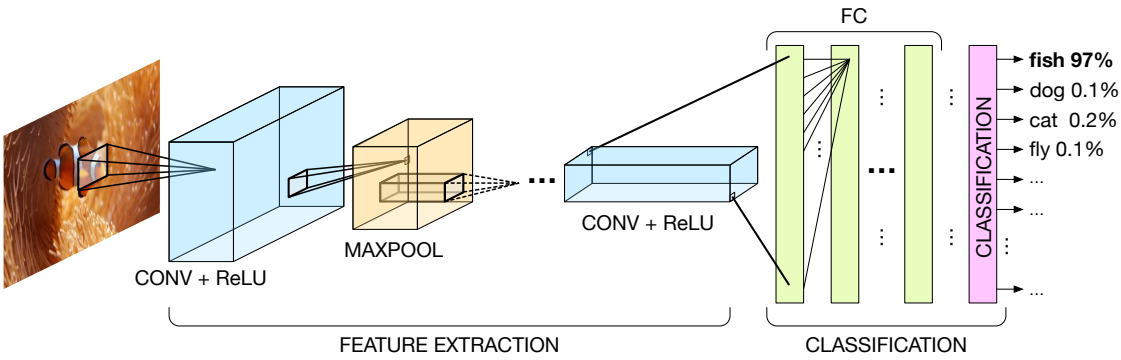

Figure 1.16: A typical CNN architecture is composed of a sequence of convolution, ReLU and max pooling layers whose job is to extract features from the input image. After feature extraction, there is a classification stage composed of a number of fully connected (FC) layers and a final classifier that computes the probability distribution over the available classes.

underline that the shape of the different layers varies across the network. The last convolution layer is fed to the so called *fully connected* (FC) layers. Differently from standard convolution layers where each neuron is connected only to a small portion of the previous layer, in FC layers all neurons are connected to the entire previous layer (that is why they are called fully connected). FC layers generate a vector whose size is equal to the number of classes that the network can discriminate. After the FC stage, the last layer inside a CNN is a classifier that returns the probability distribution over the available classes. The class with the highest probability is the result of the CNN classification. Feeding the network with new images (not the one in the test dataset) for classification is referred to as *inference* phase.

1.4 Case study: AlexNet

ILSVRC (ImageNet Large Scale Visual Recognition Challenge) is an annual competition, announced for the first time in 2010, in which teams of researchers compete with their algorithms to reach the highest precision in different visual recognition tasks. The goal of the challenge is to classify images using a subset of the

ImageNet dataset (10 million images divided in more than 10 thousands categories) as training [93]. In 2012 a deep CNN called AlexNet [69] made a breakthrough by lowering the classification error rate to 16% (until then it was around 25%). This is considered as the beginning of the deep learning revolution. Since then, not only the AI research community but also many other research fields and the industry have started paying attention. After AlexNet, several novel and powerful CNN architectures have been proposed, reaching excellent visual recognition results (the classification error rate is now smaller than 5%).

In order to fully understand some important challenges related to deep CNNs, a brief overview of AlexNet will be given in the following.
AlexNet is composed of 8 layers: 5 convolutional, and 3 fully connected layers. Every convolutional and FC layer is followed by the ReLU non-linearity and the first, the second and the fifth CONV layers are also followed by max pooling. The input are 224×224×3 images. The last FC layer is connected to a 1000-way classifier that generates a probability distribution over the 1000 classes that the network can discriminate. A detailed scheme of the architecture can be found in [69]. The main figure of interest in the contest of this work is the computational power and the storage capacity needed to run AlexNet. As highlighted in figure 1.17, this network

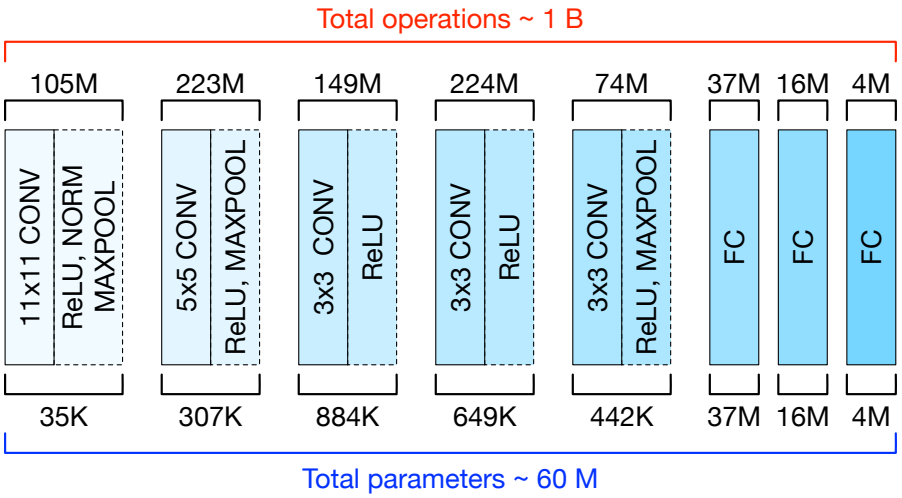

Figure 1.17: AlexNet parameters count.

has 60 million parameters and it counts almost 1 billion operations. In particular, convolutional layers account for more than 90% of the operations, being the true *computational bottleneck* of CNNs.
The time required to train AlexNet, as the authors claim, was between five and six days on two GTX 580 3GB GPUs. It is important to underline that while training requires *high-precision computation* (floating point operations), inference does not have the same precision constraints [51][30][21] and, in fact, the latency is smaller than in the training case.

1.5 Challenges of processing in CNNs

In general, deep learning algorithms must deal with the manipulation of a *huge quantity of data* and with very *long processing time*. As seen in section 1.4, AlexNet is characterized by a large quantity of parameters and operations. Over the last ten years, CNNs have evolved and become deeper and more complex. Figure 1.18 highlights how the complexity of CNNs has drastically grown over the years by taking as reference five very well known networks (LeNet [71], AlexNet [69], GoogLeNet [110], VGGNet [106], ResNet [48]). The direct consequence of this complexity growth is twofold. On one side there is the need for powerful systems that can efficiently sustain such *data-intensive* workload. For this reason, the diffusion of deep CNNs has overlapped with the appearance and consolidation of massively parallel technologies such as GPUs (Graphic Processor Units). On the other side, CNN processing is highly demanding in terms of *memory requirements* (a large number of parameters), indeed, memory access is the bottleneck as convolution operations require a lot of read/write accesses to retrieve input data and write partial results that will be reused for subsequent convolution operations (the output of a layer is the input of the next layer). The CNN data flow allows to reuse some input data. This *data reuse* property involves both kernels and feature maps.

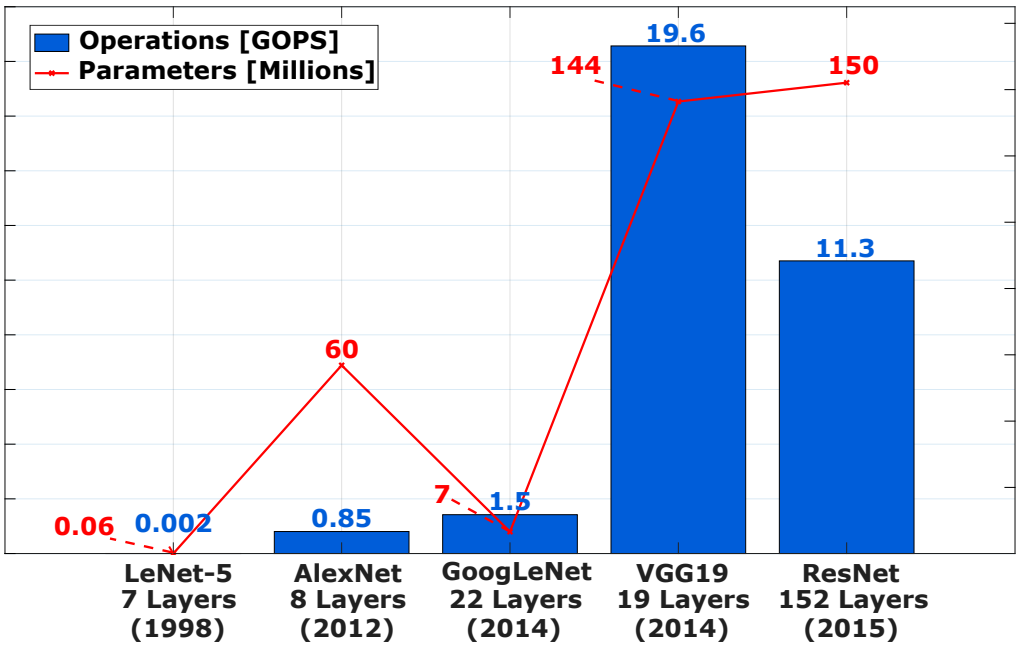

Figure 1.18: Complexity evolution of CNNs over time.

- Kernel reuse: as explained in subsection 1.3.1, because of the weight sharing property kernels are reused multiple times over an input feature map.

- Input feature map reuse: because of the sliding window process explained in subsection 1.3.2, pixels are reused across convolution windows. Moreover, the same feature map is reused across different filters (figure 1.15).

Nevertheless, memory access remains a big challenge.
The memory-demanding and computational-intensive characteristics of CNN processing bring in a further challenge: *power consumption*. As said when introducing Convolutional Neural Networks, they are effectively used in manifold applications. Many of these applications are mobile or IoT related and they make use of a pre-trained network to perform *real-time inference*. It is clear that such kind of applications have very *strict power and latency requirements*. For this reason, the research community and the industry have been focusing on energy-efficient *inference acceleration* and many hardware solutions have been proposed. An overview of the main ones will be given in the following chapter.

Chapter 2

Related Work

The State of the Art on acceleration of Convolutional Neural Networks is broad. Here only some of the most relevant works will be reported. The accelerators described in the following are all based on ASIC (Application Specific Integrated Circuit) designs as that is the target of this work (the motivation of this choice will be given at the beginning of chapter 3). An extensive survey on hardware accelerators (including FPGA- and GPU-based ones) for Deep Learning can be found in [112] (together with other related topics) and [117].

In [20] authors propose a reconfigurable accelerator for Deep CNNs that is optimized for energy efficiency. It is based on a spatial architecture made of an array of 168 processing elements (PEs) and four levels of memory hierarchy that include the off-chip DRAM, the on-chip global buffer, a FIFO buffer that controls the inter-PE communication traffic and a register file inside each PE. In addition, the PE contains a MAC for multiply-accumulate operations that can also be used for max pooling processing. They also define a taxonomy of existing CNN dataflows (each of them exploits only partially the data sharing possibilities offered by the convolution) and, based on those, propose a novel dataflow called row stationary that takes advantage of all the types of data sharing in convolutional layers to efficiently exploit and optimize the data movement across the memory hierarchy.

The work presented in [19] is an area- and energy-efficient accelerator, scalable to TOP/s performance called Origami. It is composed of a number of so called sum-of-product (SoP) units that compute convolution windows. Each SoP is fed with the same input features but with different kernels, hence, each SoP compute the partial sum of a different output channel. These partial sums are then accumulated by other blocks called channel summer (ChSum). Origami is also equipped with an SRAM that stores the image window that is currently being computed, an image bank (basically a register file) in which is loaded the image row to send to the SoP units and a filter bank for storing weights. The Origami chip is thought to be used in conjunction with an FPGA that is configured to do possible data pre-processing on input images, store data processed by Origami chips, execute max pooling and

ReLU functions that Origami does not support.

The Deep Convolutional Neural Network Recognition Processor presented in [105] is an energy-efficient CNN processor to be used in IoE (Internet-of-Everything) systems to enable in-situ machine learning processing. The system is made of four homogeneous CNN cores composed of: a control unit, input, output and kernel buffers and two so called Neuron Processing Elements (NPEs). Each NPE is composed of a number of parallel dual-range MAC blocks (DRMAC) for convolutions, ReLU blocks and Maxpool blocks. The DRMAC blocks can be configured to perform truncated MAC operations on a reduced bit-width precision to limit the power consumption. In addition the authors conduct a PCA (Principal Component Analysis) to extract few basic kernels from which the original kernels can be derived by weighted sums of such basic kernels with a very small loss in the network accuracy. The advantage of this approach is that only the basic kernels and the constant values needed for the weighted sums are stored on chip.

The work in [46] presents an energy efficient inference engine (EIE) designed for accelerating compressed network models based on sparse matrix-matrix multiplications and weight sharing without accuracy loss. The engine is composed of a central control unit (CCU) that controls an array of PEs each of them computing convolutions on a row of the compressed input matrix. The PE is composed of an arithmetic unit that performs MAC operations and various memory units to manage input/output data movements. Furthermore, there is a distributed non-zero detection network that detects non-zero input features which are then broadcasted to the PE array.

In [80] authors propose an energy-scalable CNN processor that can be used for visual recognition in wearable devices. A subword-parallel Dynamic-Voltage-Accuracy-Frequency Scaling (DVAFS) technique is introduced and exploited to enable energy-precision scalability while always guaranteeing a constant throughput. The processor is C programmable and it is based on a SIMD RISC instruction set extended with custom instructions. The processor is composed of a SIMD array of MAC units for convolutions, a SIMD array of units that perform ReLU and max pooling, an on-chip SRAM and a control unit. The MAC units are designed to reconfigurable in terms of precision: as instance, a single unit can be used to execute one MAC operation on two 8-bit data or two MAC operations on four 4-bit data. Moreover, in order to exploit dynamic voltage scaling in a granular way, the chip is divided into three different power and body-bias regions.

The work presented in [29] is a complete energy-efficient SoC (System-on-Chip) for embedded systems that make use of CNN applications. It composed of an ARM Cortex microcontroller, eight DSP clusters that perform operations such as pooling, non-linear activation, normalization and classification, 8 convolutional accelerators, DMAs units, different peripherals and other standard blocks to support different computer vision applications and on-chip SRAM. Each convolutional accelerator integrates buffers for input features, kernels, intermediate and output results, 36

fixed-point MAC units and an adder tree for accumulation.

The Deep Neural Processing Unit (DNPU) proposed in [103] is a reconfigurable processor that support CNNs and RNNs (Recurrent Neural Networks). It is composed of a convolution processor, a unit that processes FC and RNN layers and a global RISC controller. The convolution processor is a multi-cluster unit, where each cluster (four in total) integrates four convolution cores, each having an array of PEs and small memory units to hold input data and partial sums. The key feature is the use of a dynamic fixed-point with on-line adaptation technique and of a LUT-based (Lookup Table) multiplier that allows to reduce the power consumption with respect to a standard multiplier. The adaptive dynamic fixed-point changes the fraction length of a word, based on the overflow monitoring of the operations executed. The processor for FC and RNN layers is composed of buffers for input/output data and a matrix multiplier based on a quantization table (Q-table) where pre-computed multiplication results between input features and weights are stored. The Q-table allows to reduce the off-chip accesses to retrieve input data.

In [122] authors propose an energy-efficient Hybrid-Neural-Network Processor that supports CNN-, FCN (Full Connection Network)- and RNN-like dataflow. The processor is composed of a controller, an on-chip memory system and two heterogeneous arrays of processing elements. PEs are divided in general PEs for convolution support and super PEs which are enhanced general PEs that also support pooling and operations needed for RNNs. Each PE has two configurable multiplier that can be used together or separately depending on the input data precision.

Paper [63] describes the architecture and the deployment in datacenters of the Google Tensor Processing Unit (TPU). The TPU is designed to work as a coprocessor inside servers to accelerate DNN computation and meet the requirements of the growing demand of DNNs usage in applications such as speech recognition. The TPU is a custom ASIC whose main computation core is a Matrix Multiply Unit containing 256×256 MAC units organized as a systolic array. An accumulator unit accumulates the partial sums produced by the Matrix Multiply Unit. There are different on-chip buffers for storing weights, input features, partial results and instructions for the TPU coming from the host server. That are also two other units, one for calculating activations and one for performing normalization and pooling. TPU's instructions are CISC-like.

Chapter 3

Deep Learning Processor Architecture

This chapter thoroughly describes the architecture of the proposed Deep Learning Processor. The accelerator has been designed having in mind the challenges that CNN processing poses, as discussed in chapter 1, section 1.5. This part of the work, together with the design space exploration of the DLP (chapter 4), was developed in the contest of an international joint research project involving Politecnico di Torino (Giulia Santoro, prof. Mario Casu, prof. Andrea Calimera and Valentino Peluso) and National University of Singapore (prof. Massimo Alioto). The outcome of this work is the product of the discussion and the collaboration between the people cited above. Part of this work was previously published in [98] and [97].

3.1 Hardware Choice for Inference Acceleration

Inference acceleration of CNNs has been widely addressed in literature (chapter 2) and many different hardware solutions have been proposed.
The key features that an optimal DL accelerator should have are:

- *flexibility* to support the shape variability of CNN layers;
- *high-throughput* to keep up with the computation-intensive workload;
- *energy-efficiency* to sustain the throughput and the data demand at the same time.

Given the peculiar characteristics of CNNs, not all hardware systems are suitable for inference acceleration. Table 3.1 reports an overview of advantages and disadvantages of using some common hardware solutions.
CPUs are flexible since they are general purpose but they are highly inefficient for

HW type	PROs	CONs
CPU	Flexible	Hihgly inefficient
GPU	Great for training	High power dissipation
FPGA	Reconfigurable Not expensive	Not dedicated
ASIC	Dedicated Performance	Not flexible

Table 3.1: Advantages and drawbacks of different types of hardware solutions for inference acceleration.

data-intensive and massively parallel applications such as deep learning. GPUs, on the contrary, have been and are still used for accelerating the training phase of CNNs because they provide high-precision computation (required for training) and massive parallelism. However, GPUs are characterized by a huge power consumption that makes them unfit for energy efficient inference that is instead required by most of the emerging applications, such as embedded platforms. Many hardware accelerators in literature are based on FPGA (Field Programmable Gate Array) implementations since they are a low cost solution while also ensuring reconfigurability. However, they are not a dedicated solution, hence they cannot reach the same performance that an ASIC implementation would. Clearly, the drawback in ASIC solutions is that they are not flexible.
An optimal trade-off between flexibility, performance and efficiency is represented by a hybrid solutions such as a *Deep Learning Processor* (DLP) which combines the quality of design achieved with ASIC implementation flows with the reconfigurability of FPGAs.

3.2 System Overview

As shown in figure 3.1, the proposed DLP (hereinafter also called accelerator) is an array of Processing Elements (PEs) interconnected through a wormhole-switching Network-on-Chip (NoC) [13]. The NoC was not developed as part of this work and, since it is not the main focus, it will not be described further. The PEs array is also interfaced with an external DRAM through DDR channels. Moreover, there is an SRAM buffer that works as an additional level of memory hierarchy between the NoC, that routes data from/to the PEs array, and the off-chip DRAM. The computing core of the DLP is the array of Processing Elements that process data in parallel. Each PE communicates with the NoC through a router (small green block denoted by R) that sends data and instructions to be executed. As

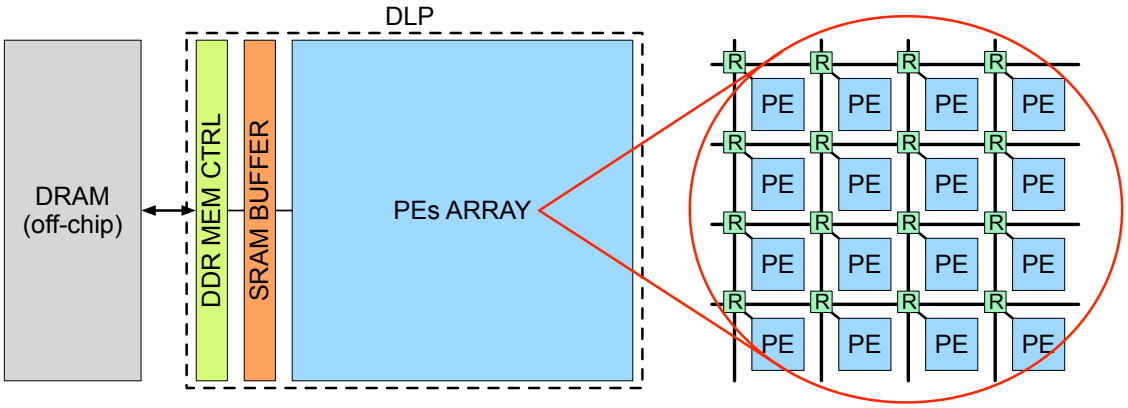

Figure 3.1: Overview of the proposed DLP architecture.

it will be thoroughly described in the next section, each PE is equipped with an input buffer to hold data to be elaborated and with a small output buffer to keep computed data that are then sent out of the array.

3.3 Processing Element

The Processing Element is where the computation happens, hence, it can be considered as the basic block of the whole accelerator.
It is a micro-programmed Single Instruction Multiple Data (SIMD) unit, as depicted in figure 3.2. It is composed of a *Control Unit* (CU) that sends instructions to the *SIMD unit*, called so because it is made of four *lanes*, each executing the same instructions but on different data. Lanes receive data from the router and send back computed data.
The Processing Element has been designed to be:

- optimized for CNN-like workload
- programmable and flexible
- performance-oriented

The PE was described in VHDL (VHSIC Hardware Description Language) in a parametric fashion with maximum flexibility. This means that every characteristic (e.g. number of lanes, data precision, size of the PE internal memory and so on) can be easily modified without need to change the code.

3.3.1 Lane

As depicted in figure 3.2, each lane is composed of:

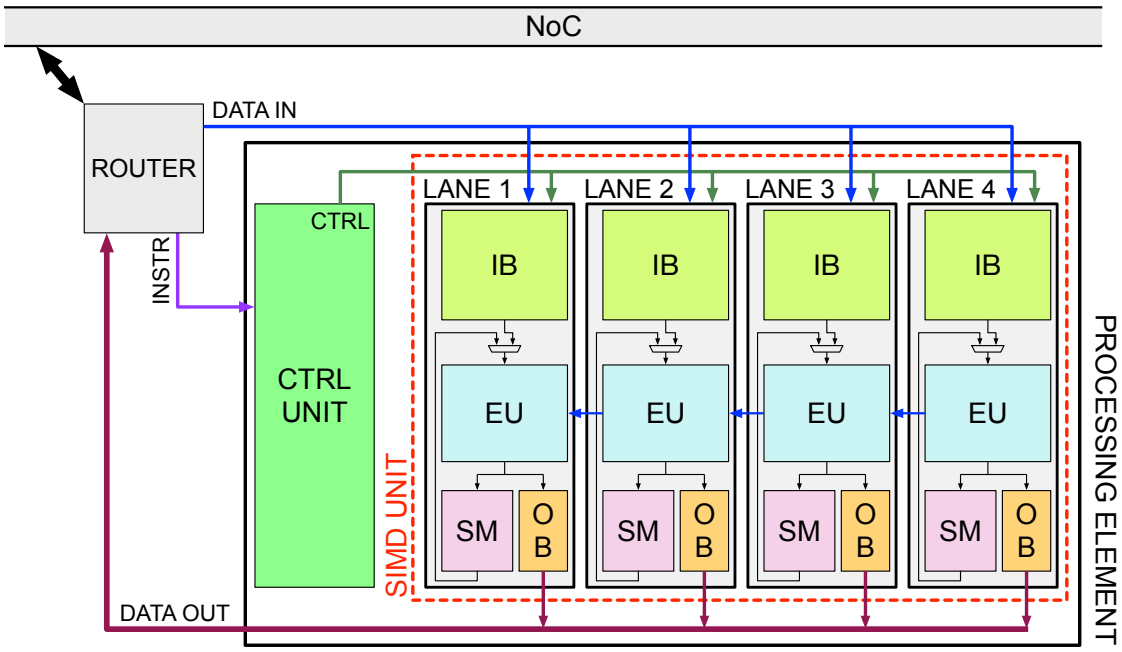

Figure 3.2: Block diagram of the PE architecture.

- an Input Buffer (IB) that stores data to be computed;
- an Execution Unit (EU);
- a Scratchpad Memory (SM) that holds partial results that will be reused shortly afterwards;
- an Output Buffer (OB) to buffer computed data that are then sent out of the PE.

Looking more closely at figure 3.2, it can be noticed that lanes are connected with each other (blue arrows going from an EU to the other). In fact, in addition to working independently, lanes can also cooperate. This *cooperation mechanism* can be applied mainly in two cases.

1. When dealing with large-sized input data that cannot be handled by a single lane, the data is split in several chunks that are assigned to different lanes. Each lane executes a portion of the operation and then partial results are properly elaborated exploiting the interconnection among the EUs to produce the final result.

2. When accumulating data to compute the final result of a convolution window: as shown in figure 3.3, each lane computes the multiplication of an input pixel and a weight. The partial results produced are then accumulated horizontally through the lanes to obtain the final result of the convolution window.

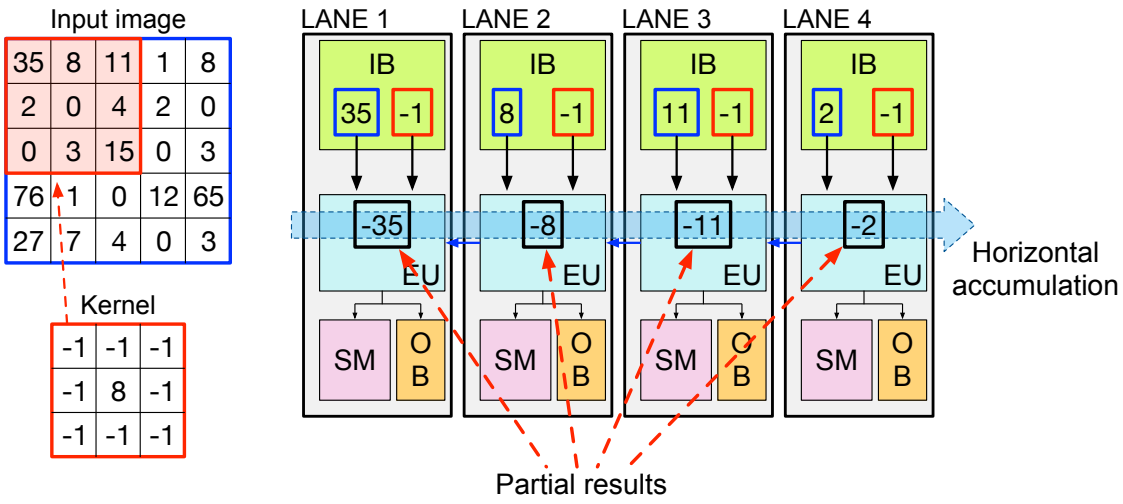

Figure 3.3: PE lanes working in group.

This grouping mechanism makes it possible to *save data movement* from/to memory. Lanes can be grouped to work together in different combinations: as instance, two groups of two lanes each or one group of four lanes. Some lanes can also be left in an idle state if they are not needed. The working mode of lanes inside a PE is flexible depending on the needs.

Execution Unit

The Execution Unit is a two-stage-pipeline modified Multiply-Accumulate (MAC) unit that can support multiply-accumulate operations needed for the convolution, max pooling and ReLU. These operations are the most commonly used in CNNs, hence, they have been chosen as basic operations that each lane can execute. A simplified block diagram of the EU is depicted in figure 3.4. Inputs A and B (a pixel and a weight) are used for convolution operations and come from the input buffer. As a first step they are multiplied, then this partial result is accumulated through the adder. A multiplication and an accumulation take two clock cycles because of the presence of two registers (REG and ACC) that define the two pipeline stages of the EU. At each clock cycle new A and B inputs are loaded from the input buffer, then they are multiplied and after that summed to the previous partial result until a convolution window is completed. Once the final result is available, the EU stores it in the output buffer or in the scratchpad memory if it will be needed again shortly afterwards.
Multiplexer M1 is used to select between:

① data coming from the multiplier;

② input D that may come from the IB or from another lane. This input is

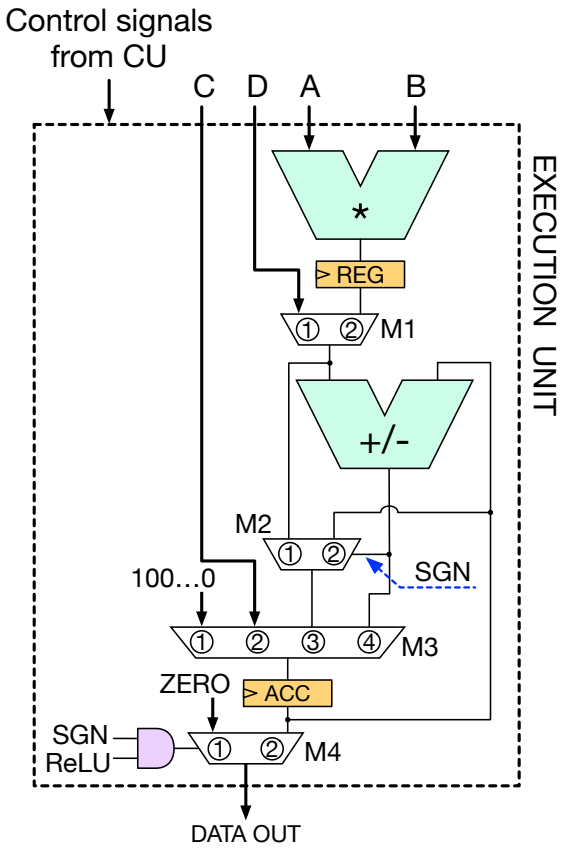

Figure 3.4: Simplified block diagram of the Execution Unit.

selected when the adder is used as a standalone component to sum partial results (e.g., during cross-channel accumulation in high dimensional convolution or for horizontal accumulation across lanes) or for max pooling.

Multiplexer M2 is used to support max pooling. The EU compares two data at a time by subtracting them. The sign (SGN) of the result is used to select the bigger data between the two inputs of the adder that might be:

① the data coming from multiplexer M1;

② the value stored in the accumulation (ACC) register.

The comparison is carried on until all data in a max pooling window have been compared and the biggest among them is chosen.
Depending on the operation to perform, the ACC register must be properly initialized. Multiplexer M3 is used for this purpose and it selects among:

① the smallest negative value when performing the first iteration of max pooling;

② the input C coming from the SM;

③ the output of multiplexer M2 which contains the bigger value among the two compared when performing max pooling;

④ the output of the adder.

Finally, multiplexer M4 is used when the CONV layer that is being computed is followed by the ReLU activation. As said in paragraph 1.3.2, a ReLU activation saturates to zero negative values. As a consequence, if the ReLU signal is high and the value coming from the adder is negative (SGN equal to 1), the output of the AND will be high as well and it will select input ① of the multiplexer.
The execution unit is very flexible and with minor modifications with respect to a standard MAC unit it provides hardware support not only for convolution but also for ReLU, max pooling and accumulation of partial results that might be stored, depending on the case, either in the scratchpad memory or in the input buffer or that come from other lanes inside the PE.

Input Buffer

The input buffer is a sophisticated and flexible memory structure. A simplified block diagram is depicted in picture 3.5. The IB is composed of two banks: a private (PVT) and a shared (SHD) one. The PVT bank, as the name suggests, is only accessible by its lane. The SHD bank, instead, might be accessed also by the other lanes in the PE. This *sharing property* is useful when lanes share the same kernel. If there was no sharing, it would have been necessary to replicate the kernel in the input buffer of each lane creating redundancy. By giving to each lane access to the shared bank of the other lanes, there is no such waste and all the memory capacity can be exploited to store useful data. The sharing mode can be enabled or disabled depending on the needs.
Both the PVT and SHD banks are divided in an even and an odd portion. Each portion has one write port (data_in1 for even, data_in2 for odd) controlled by a write address (wr1_addr for even, wr2_addr for odd). In total, each bank has two write ports. For what concerns the read ports the situation is different. The PVT bank has four read ports (two per portion, one controlled by rd1_addr and the other by rd2_addr). The SHD bank has, instead, two read ports (both controlled by rd1_addr).
Multiplexers are used to control from which bank and portion of the IB forwarding the data to output ports. As indicated by the dashed red arrows, multiplexers are controlled by read addresses (actually, only some bits of the whole addresses are used as control signals). Thanks to these multiplexers, different read combinations are available guaranteeing flexibility:

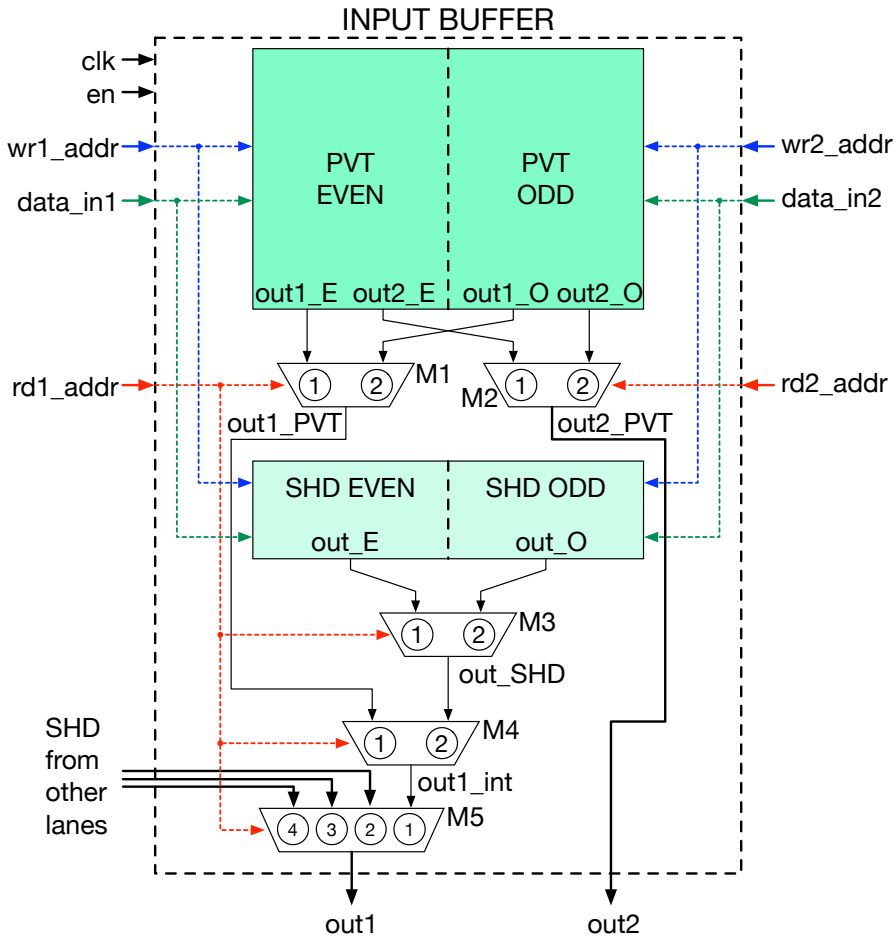

Figure 3.5: Simplified block diagram of the Input Buffer.

- both out1 and out2 from PVT (one from the even and the other from the odd portion);
- out1 from local SHD and out2 from PVT;
- out1 from other lane SHD and out2 from PVT.

The first read option might be useful, as instance, when performing max pooling: if the PVT bank contains data to be compared, they are read two at a time and sent to the EU. The last two options are useful for convolution operations: supposing that PVT contains pixels and SHD stores weights, one data is read from PVT and the other from SHD.
In order to understand how the address space of the input buffer is managed, it might be of help to report an example. Suppose that each lane has an IB with 160 locations (128 for the PVT bank and 32 for the SHD) as shown in figure 3.6. In

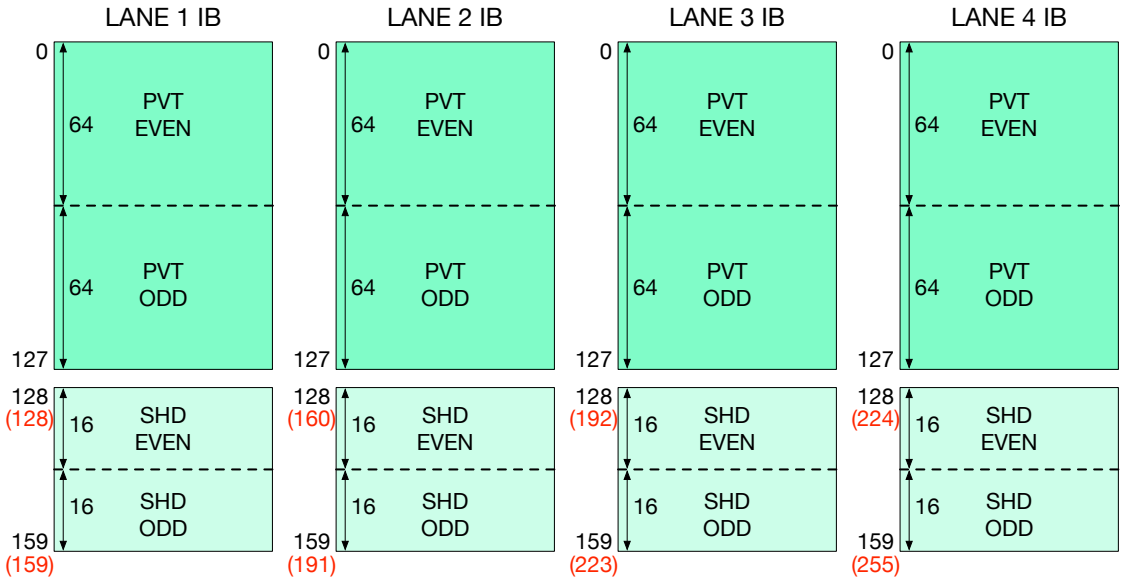

Figure 3.6: Address space of the Input Buffer memory system inside a PE.

addition, all the four SHD banks, for a total of 128 locations, are addressable by each lane. This means that each lane can address a total of 256 locations, requiring an 8-bit address. Two cases must be distinguished.

1. Sharing mode disabled (each lane has not access to the other SHD banks): the address space seen by each lane goes from address 0 to address 159 (0-127 for PVT and 128-159 for local SHD).

2. Sharing mode enabled (each lane can access all the other SHD banks): the address space seen by each lane goes from address 0 to address 255 (0-127 always for PVT). Differently from the first case, now each lane sees an address space for accessing all SHD banks that goes from address 128 to 255 (red numbers in figure 3.6).

The address mapping for reading/writing operations is explained in the following. SHD read address mapping:

usage	1	SHD bank sel		bank address			even/odd
bits	7	6	5	4	...	1	0

Table 3.2: Read address mapping for SHD banks.

The most significant bit (bit number 7) is set to one as SHD addresses start from 128 (10000000 in binary). Bits 5 and 6 are used to discriminate SHD banks (00 for lane 1 SHD, 01 for lane 2 and so on). Bits 0 to 4 are used to address a location in

the bank.
PVT read address mapping:

usage	0	bank address			even/odd
bits	7	6	...	1	0

Table 3.3: Read address mapping for PVT banks.

In this case the most significant bit is set to zero as PVT addresses go from 0 (00000000 in binary) to 127 (01111111 in binary).
SHD write address mapping:

usage	1	unused		bank address			even/odd
bits	7	6	5	4	...	1	0

Table 3.4: Write address mapping for SHD banks.

In this case bits 5 and 6 are unused since the writing is managed privately inside each lane (there is no need to share as done for the reading).
PVT write address mapping: Another fundamental characteristics of the IB is that

usage	0	bank address			even/odd
bits	7	6	...	1	0

Table 3.5: Write address mapping for PVT banks.

is can be used in *double buffering* mode (DBM). When executing convolutions, as instance, data are continuously read from the IB. Once all of them have been consumed by the EU, the computation must stop allowing new data to be loaded in the input buffer. This loading latency causes performance to decrease. In order to avoid it, the double buffer mode was introduced. Basically, the IB is divided into two parts, one read portion and one write portion as shown in picture 3.7. Assume that, at step 1, the orange portion is being loaded with new data while the green portion is being read. At step 2 all data from the read portion has been used to feed the EU while the data loading of the write part is terminated. At this point the two portions are swapped: the green one becomes the write portion while the orange one is used to read the new data just loaded. Once all data in the orange block are consumed, the two portions are swapped again and so forth. Assuming that the loading and consuming rates are the same, this mechanism ensures:

- *masking* of the loading latency of the input buffer;
- *continuous computation.*

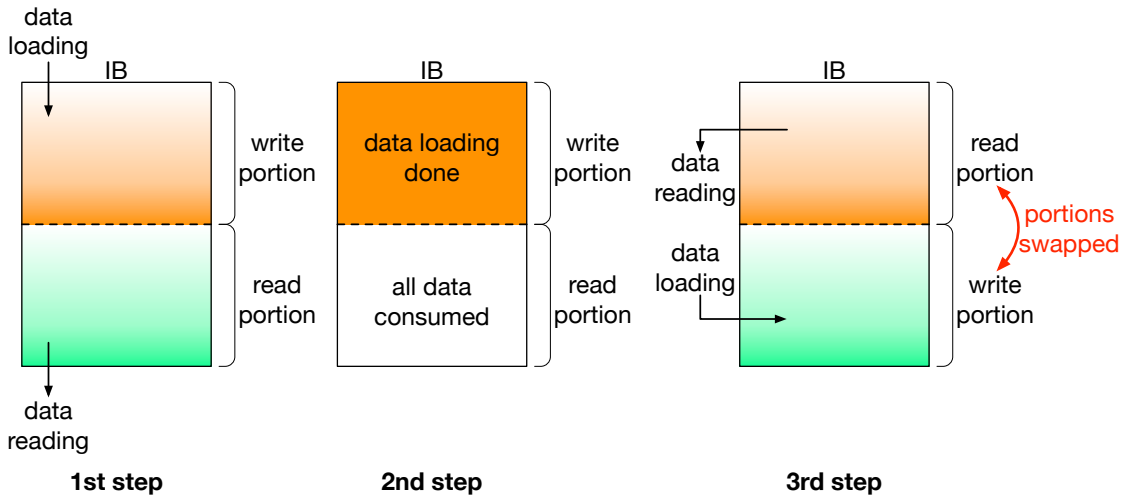

Figure 3.7: Double buffering mode of the IB.

For the sake of flexibility, the double buffering mode can be applied to PVT and SHD banks independently. Therefore, the possible combinations are:

1. no DBM for PVT and SHD;
2. DBM only for PVT or the opposite;
3. DBM for both PVT and SHD.

Scratchpad Memory

The scratchpad memory is a basic read/write memory with one input port and one output port. A scheme is shown in figure 3.8.

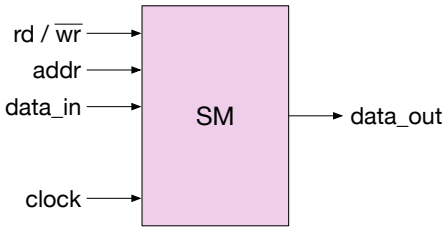

Figure 3.8: Scratchpad Memory.

Output Buffer

The output buffer depicted in figure 3.9 is a simple FIFO (First In First Out), hence, the first data stored is also the first to be read. Empty and full signals are

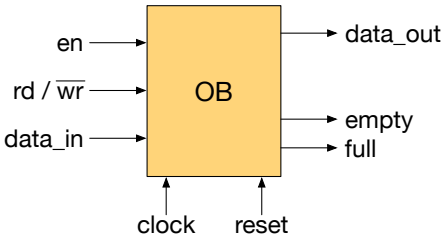

Figure 3.9: Output Buffer.

used to manage read/write operations. Whenever the buffer is full, the full signal is high and any further write operation is blocked (disabling the OB through the en signal) until data inside the buffer are read and sent to the router. Once the buffer is emptied, the full signal goes low, the empty signal goes high and write operations are enabled again.

3.3.2 Latch-based Memory Design

All memory structures described since now have been designed following a latch-based approach as the one depicted in figure 3.10. Memory cells are high-level

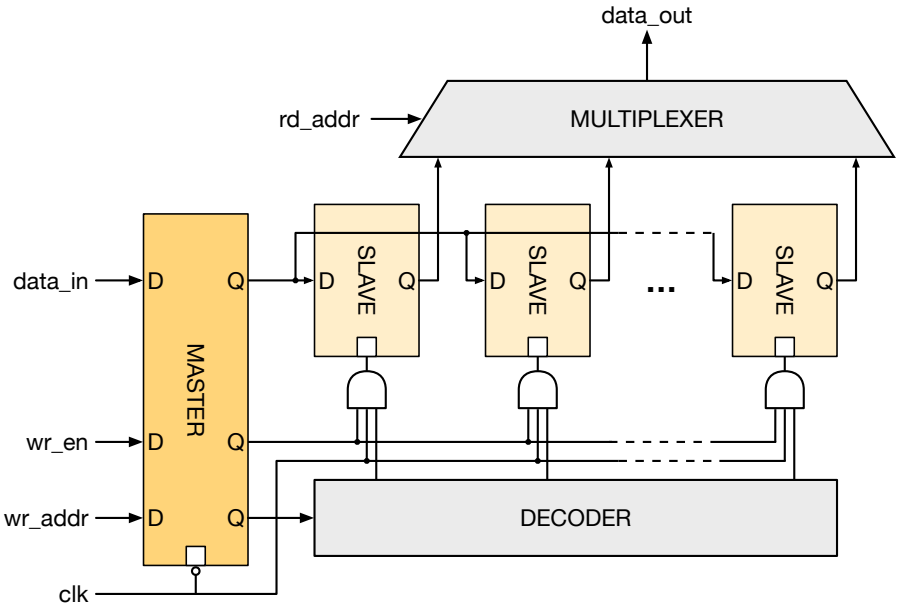

Figure 3.10: Latch-based memory design.

sensitive latches. All input signals (input data, address, ...), before entering the latch array, go through low-level sensitive master latches. Hence, there is a sequence

of master and slaves latches and, as a result, the whole memory array is edge-sensitive. This design technique is used for implementing small embedded memories (as this is the case). In fact, the presence of several slave latches controlled just by one master latch allows to build compact memory arrays. In addition, with respect to a standard SRAM, the latch-based implementation can work at a lower V_{DD}, with consequent power savings. Moreover, a latch-based memory design is a good candidate for custom of semi-custom implementation (place&route, as instance).

3.3.3 Control Unit

The control unit manages all operation modes of the PE. It is a micro-programmed unit and its block diagram is represented in picture 3.11. In addition to the standard

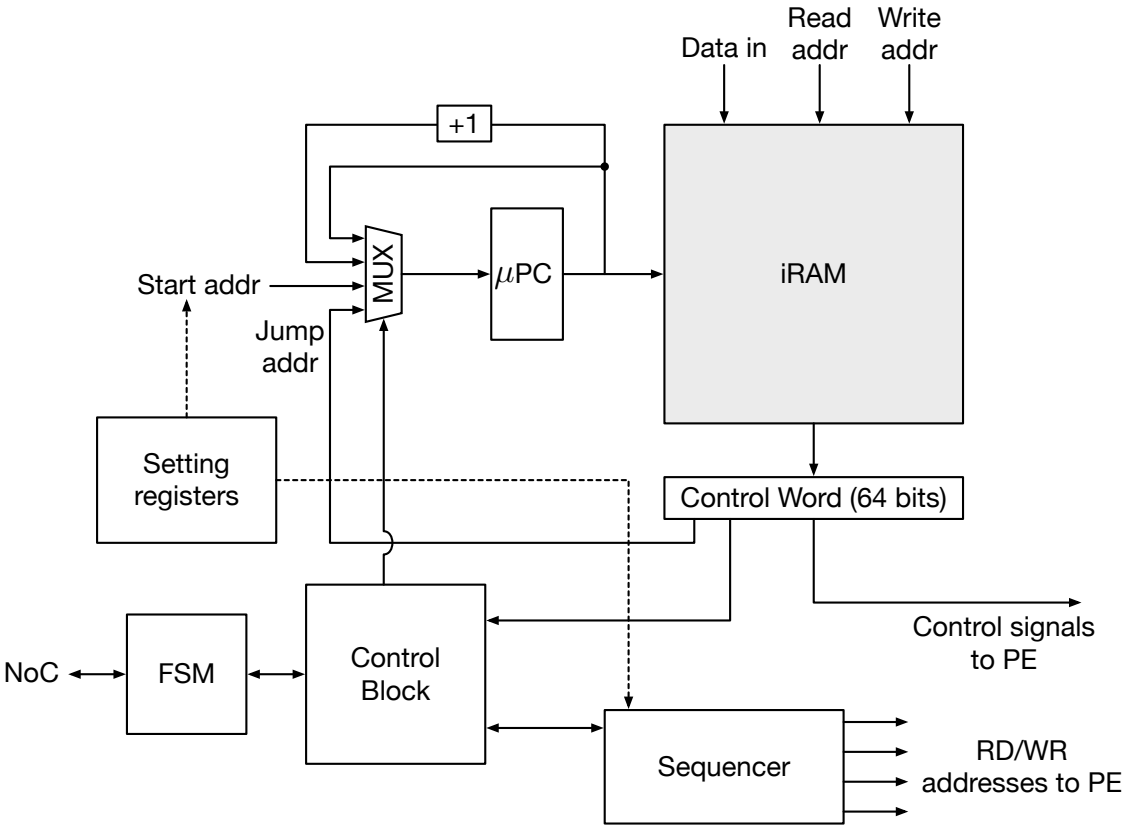

Figure 3.11: Control Unit.

blocks usually present in a micro-programmed CU, there are three additional components: the FSM (Finite State Machine), the sequencer and the setting registers. The FSM is the only component inside a PE that interacts with the external world. In particular, the FSM communicates with an interface block, which intermediates between the PE and the NoC, using request and acknowledgment signals. This

communication protocol is used to manage data transfer to/from the PE (write code or data in the PE or transfer computed data out of the PE).
The setting registers, as the name suggests, store *important settings* used to manage PE operation modes.
Finally, the sequencer (refer to figure 3.12) is a block that allows to manage instruction loops efficiently, by guaranteeing *zero-overhead in loops*. In fact, a con-

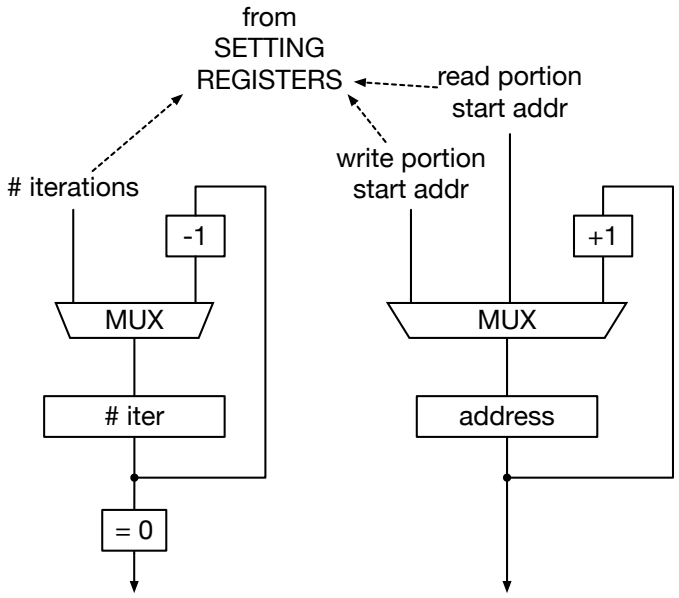

Figure 3.12: Simplified version of the sequencer.

volutional layer can be seen as a routine of few simple instructions iterated many times. The sequencer provides:

- control for iteration on a single instruction or a routine;
- automatic update of read/write addresses for the IB when reading/writing on contiguous locations;
- automatic update of addresses when using the IB in double buffering mode to swap between read/write portions.

Inputs of the sequencer come from the setting registers.

Chapter 4

Design Space Exploration

The Deep Learning Processor described in chapter 3 is used as an architectural template for conducting a design space exploration based on an analytic model that takes into account all the key features of the accelerator and defines all the best DLP configurations in terms of energy efficiency and throughput.

4.1 Low-power Design

The PE and the NoC are developed at register-transfer level using a fully parametric HDL (Hardware Description Language) code that can be easily deployed on an FPGA or mapped onto an ASIC target technology. In this work a commercial 28 nm UTBB FDSOI technology is used. The RTL (Register-Transfer Level) code of the PE and the NoC router are first synthesized using Synopsys Design Compiler, and then placed and routed using Synopsys IC Compiler. For both the design stages low-power optimization features have been enabled. Power consumption is extracted using Synopsys Prime-Time with SAIF back-annotation. Figure 4.1 shows the design flow. The FDSOI technology is well-known for its extended

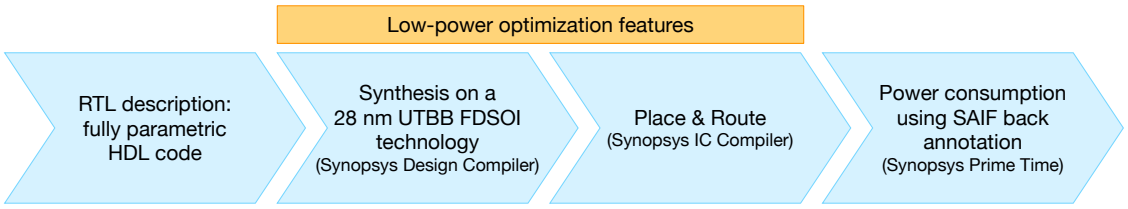

Figure 4.1: Low-power design flow of the PE.

supply voltage (V_{DD}) range. In this work, the PE and the NoC router are characterized using a V_{DD} voltage range between 0.6 V and 1.0 V with a 100 mV step and a corresponding set of clock frequencies (f_{ck}) determined after physical design. In addition, since clock frequency can be set with a finer granularity than the supply

voltage (PLLs can be tuned in more easily than voltage regulators), it has been considered also the possibility of scaling down the clock frequency for each voltage value. This results in four different frequencies for each 100 mV interval. Figure 4.2 shows the resulting seventeen Voltage-Frequency (VF) points used during the design space exploration.

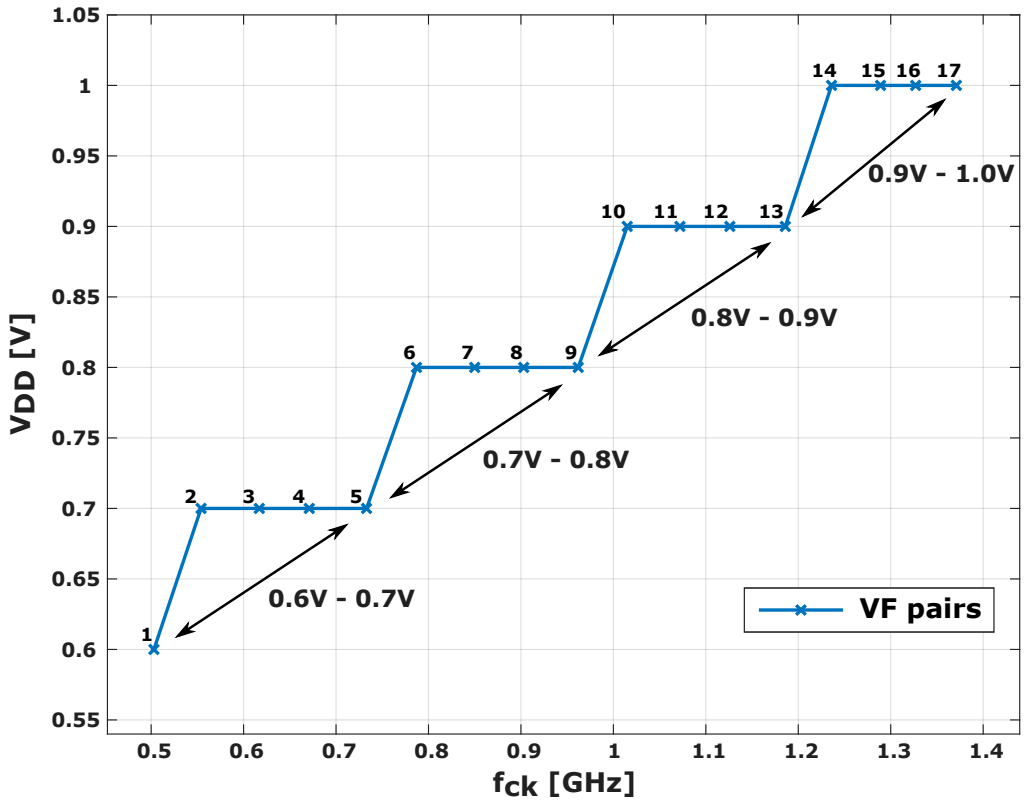

Figure 4.2: VF pairs.

4.2 Power Management

When dealing with low power and energy efficient applications, a power management strategy is crucial. Previous works, such as [22], have shown that techniques like Dynamic Voltage and Frequency Scaling (DVFS) help improve the power consumption in CNN accelerators. In this contest, a fine dual-V_{DD} DVFS management has been explored.
Standard DVFS adjusts both supply voltage and clock frequency to reduce the power consumption, exploiting the dependence of dynamic power on V_{DD} and f_{ck}. Inside a voltage range, there are different admitted clock frequencies (as shown in figure 4.2). A reduction of f_{ck} implies a linear decrease of the power consumption.

When f_{ck} is lowered below the lowest value in that voltage domain, V_{DD} is also decreased causing a further power reduction.
When considering dual-V_{DD} DVFS, the design is divided in regions called *tiles* that can be supplied with a low or a high-V_{DD}. These two V_{DD} correspond to the outermost supply voltage points inside a voltage domain (for example, 0.6 V and 0.7 V, observing figure 4.2). The V_{DD} value assigned to each tile is such that the clock frequency constraint is met and the total number of tiles set at low-V_{DD} is maximized. The dual-V_{DD} scheme assigns to tiles that belong to critical timing paths a high-V_{DD} and to tiles that belong to non-critical timing paths a low-V_{DD}.
The dual-V_{DD} DVFS technique was applied in the contest of this work by following the same implementation flow proposed in [83].

4.3 Design Space

As already said, the DLP architecture presented in chapter 3 was described using a parametric HDL code, hence, it can be used as a template to generate different instances whose characteristics vary according to table 4.1. So, each of

Variable	Value	Meaning
$\boldsymbol{N_{PE}}$	4÷100	Number of PEs
$\boldsymbol{N_{MC}}$	1÷4	Number of memory controllers
$\boldsymbol{S_{\%}}$	0÷80%, 10-% steps	Size of SRAM buffers as a percentage of die area
$\boldsymbol{B_{NoC}}$	32÷256, 32-bit steps	NoC channels bit-width
$\boldsymbol{M_{IB}}$	32, 64, 128, 256	Words in the IB
$\boldsymbol{IF_{IB}}$	8, 16, 32	Words in the IB dedicated to IFs
$\boldsymbol{DB_{IF}}$	0, 1	Double-buffering mode variable for IFs
$\boldsymbol{DB_{W}}$	0, 1	Double-buffering mode variable for Ws
$\boldsymbol{VF}$	1÷17	Voltage-Frequency pairs

Table 4.1: DLP's design space variables.

these DLP instances can vary in terms of number of PEs in the array, number and size of SRAM banks connected to DDR memory controllers and/or to the periphery of the NoC, size of input buffers and whether or not the double buffering

mode is active (this mode is set at firmware level in the microcode memory of the control unit). The design space is then obtained by the Cartesian product of all the design-time and run-time variables shown in table 4.1 that will be delineated in the following.

- $N_{PE} = N_{PEr} \times N_{PEc}$: number of PEs in the array matching the layout of a NoC mesh. It is expressed as the number of PEs per row (N_{PEr}) multiplied by the number of PEs per column (N_{PEc}). N_{PEr} and N_{PEc} are independent variables. The maximum value that N_{PE} can assume depends on the target die area which ranges from 2 mm^2 up to 10 mm^2. In the former case $N_{PE,MAX}$ is 20, while in the latter it is 100.

- N_{MC}: number of DRAM memory controllers and channels which can vary from 1 to 4. The more the channels the higher the memory bandwidth but, also, the higher the power dissipated in the external DRAM.

- $S_{\%}$: size of SRAM buffers expressed as a percentage of the maximum die area. This variable ranges from 0 to 80%, with 10% steps. The SRAM is placed between the NoC periphery and the DDR memory controllers and it is partitioned in N_{MC} buffers. Each buffer is divided in different banks in order to ensure the same access bandwidth of the NoC. The bank size depends on $S_{\%}$, N_{MC} and the die area.

- B_{NoC}: NoC channels bit-width that varies from 32 to 256 bits.

- M_{IB}: private bank of the input buffer expressed as number of words. This variable ranges from 32 to 256 words. The size of the shared bank (SHD) is always 1/4 of the private one.

- IF_{IB}: portion of the input buffer in which Input Features (IFs) are stored. Possible values are 8, 16 or 32 words. This portion and the weights portion are eventually doubled when the double buffering mode is active and they sum up to M_{IB} words (256 maximum, in total).

- DB_{IF}: boolean variable that indicates whether the double buffering mode is active (0 for inactive, 1 for active) for the input buffer portion that stores IFs (PVT).

- DB_W: boolean variable that indicates whether the double buffering mode is active (0 for inactive, 1 for active) for the input buffer portion that stores Weights (SHD).

- VF: number indicating the VF pair chosen for a particular DLP instance.

When conducting the design space exploration, three constraints are set and feasible solution must respect them.

1. A_{MAX}: maximum die area which is given by the sum of the logic are and the SRAM area.

2. BW_{DRAM}: maximum DRAM bandwidth which is given by the multiplication between accesses per unit time to the DDR channels and the DDR channels bit-width.

3. BW_{SRAM}: maximum SRAM bandwidth (similar limit as for BW_{DRAM}).

Once all non-feasible solutions are pruned out of the design space, a Pareto analysis is conducted by searching feasible implementations which are not dominated in terms of *throughput*, evaluated as billions of basic operations per second (GOPS), or *energy efficiency*, evaluated as the inverse of the energy spent per single operation (in nJ^{-1}, commonly expressed as GOPS/W). The next section describes the analytic model in detail.

4.4 Energy-Throughput Model

4.4.1 Workload Model

The computation of CONV and FC layers is modeled as matrix-matrix multiplications as done in standard CNNs frameworks such as Caffe [61]. Only convolutional and fully connected layers are considered in the workload model as they represent the true computational bottleneck of CNNs. Standard matrix block partitioning is then applied to enable parallel processing. Hence, the computation inside each PE is modeled as a sequence of three basic steps:

1. fetching of Input Features (IFs) and Weights (Ws);

2. multiplication of the fetched IF and W;

3. accumulation.

IFs are firstly fetched from the off-chip DRAM and then stored in the SRAM buffers, if present, because they are reused across different kernels. If SRAM buffers are not present, input data have to be fetched from the DRAM whenever it is necessary.

4.4.2 Throughput Model

Different combination of design variables may lead to architectures in which the throughput can be limited by:

1. computation;

2. memory bandwidth of the DRAM of the SRAM;

3. the NoC bandwidth.

This model finds the bottleneck and computes the throughput accordingly. The throughput model has been described in such a way to follow quite accurately the RTL description of the PE. The NoC is instead modeled by following a simplified steady-state model that neglects the insurgence of congestion in the NoC and the consequent latencies. In fact, since the NoC communication patterns are regular and predictable at design-time for the application considered in this work, neglecting the congestion does not affect the accuracy of the model.
The throughput of the all DLP can be calculated as:

$$Th_S = N_{PE} Th_{PE} \tag{4.1}$$

where N_{PE} is the number of active PEs and Th_{PE} is the throughput of a single PE expressed by equation 4.2.

$$Th_{PE} = N_L Th_L \tag{4.2}$$

N_L is the number of lanes inside a PE (this is fixed to 4), while Th_L is the single lane throughput. Th_L can be seen as the number of basic operations executed by a lane during an *Initiation Interval*.
Th_L can be expressed as:

$$Th_L = \frac{N_W N_{OPS}}{II_W} f_{ck} \tag{4.3}$$

Considering that $N_W N_{OPS} = N_W(N_{MUL} + N_{ADD}) = 2N_W IF$ is the number of operations computed by a single lane during an initiation interval II_W, the expression of the throughput can be rewritten as:

$$Th_L = \frac{2N_W IF}{II_W} f_{ck} \tag{4.4}$$

As depicted in figure 4.3, the computation of a lane consists of two nested loops (loop1 and loop2). First, in loop1, weights are loaded from the DRAM into the input buffer (W portion), then, in loop2 input features are loaded from the DRAM into the IB (IF portion). The inner loop is repeated for N_W times, where N_W is the weight-reuse factor and it takes into account the fact that weights are reused over a feature map. So, except for the first iteration, for the other loop2 iterations IFs are fetched directly from the input buffer and sent to the execution unit (EU) together with the weight fetched in the outer loop. The initiation interval of loop1 and loop2 are indicated by II_W and II_{IF}, respectively. If there are no stalls due to the unavailability of a double buffer ($DB_W = 0$) or the NoC bandwidth, then the outer cycle initiation interval is exactly equal to $N_W \times II_W$.
II_W is calculated as:

$$II_W = \begin{cases} \max(L_W, N_W II_{IF}), & \text{if } DB_W = 1 \\ L_W + N_W II_{IF} & \text{if } DB_W = 0 \end{cases} \tag{4.5}$$

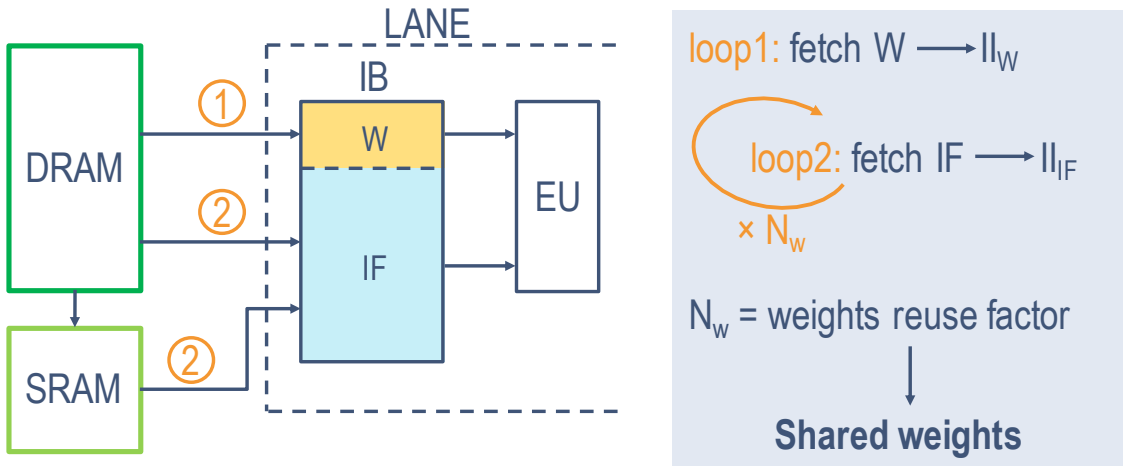

Figure 4.3: PE data fetching scheme for convolution computation.

where L_W is the weights communication latency (time needed to load the wights). The expression of II_W takes into account two cases.

1. DB_W active: II_W is equal to the maximum between the weights communication latency L_W and N_W times the IF initiation interval (II_{IF} must be taken into account for every iteration of inner loop2, as shown in figure 4.3).

2. DB_W inactive: II_W is equal to the sum of the weights communication latency L_W and N_W times the IF initiation interval.

L_W is limited by the bandwidth of either the PE input interface, the NoC, or the DRAM:

$$L_W = \frac{WD_W N_L}{\min(BW_{PE}, BW_{NoC}, BW_{DRAM})} \tag{4.6}$$

In equation 4.6, W is the number of weights loaded in the IB, D_W the weight bit-width parallelism and N_L is the number of lanes.
II_{IF} is calculated similarly to II_W; it takes into account potential stalls due to IF communication latency and computation latency:

$$II_{IF} = \begin{cases} \max(L_{IF}, L_C), & \text{if } DB_{IF} = 1 \\ L_{IF} + L_C & \text{if } DB_{IF} = 0 \end{cases} \tag{4.7}$$

where L_{IF} is the communication latency of input features, which is similar to equation 4.6 but it considers the possibility of loading IFs from the SRAM other than the DRAM. The expression of II_{IF} also takes into account two cases.

1. DB_{IF} active: II_{IF} is equal to the maximum between the IF communication latency L_{IF} and the computation latency L_C.

2. DB_{IF} inactive: II_{IF} is equal to the sum of the IF communication latency L_{IF} and the computation latency L_C.

The computation latency is calculated as:

$$L_C = IF + L_{ovh} \tag{4.8}$$

where L_{ovh} is the control logic overhead (one clock cycle in this architecture).
In a scenario where the computation latency is the dominant term, $II_W \simeq N_W II_{IF}$, $II_{IF} \simeq L_C$ and $L_C \simeq IF$ (negligible overhead L_{ovh}). When substituting these expressions in equation 4.4, Th_L becomes:

$$Th_L \simeq \frac{2IF}{II_{IF}} f_{ck} \simeq \frac{2IF}{IF} f_{ck} \simeq 2f_{ck} \tag{4.9}$$

Basically, Th_L simplifies to two operations per clock cycle, i.e. one multiplication and one addition.
The throughput of a lane is given as number of clock cycles, hence, the actual Th_L in GOPS is finally obtained by multiplying the number of clock cycles by the clock frequency f_{ck}.

4.4.3 Energy Model

The energy model takes into account the energy dissipated:

- to access both the off-chip DRAM and the SRAM;
- to move data through the NoC;
- by the PE array.

The energy efficiency (GOPS/W) is calculated as:

$$E_{eff} = \frac{Th_S}{E_{TOT}} \tag{4.10}$$

where E_{TOT} is the sum of the three energy contribution cited above and Th_S is the system throughput (equation 4.1). The power dissipated to access the DRAM considers a specific commercial low-power DDR chip (Micron Technology Inc. LPDDR4X, Z00M 4-Gb die) and its expression is taken from the datasheet (in mW):

$$P_{DRAM} = N_{MC}\Big(10.9 + 0.8\frac{522BW_{rd} + 462BW_{wr}}{BW_{DRAM}}\Big), \tag{4.11}$$

BW_{rd} and BW_{wr} are the reading and writing bandwidth, respectively, and they depend on the workload. The bandwidth of each DRAM channel is equal to BW_{DRAM}/N_{MC} =96 Gbps. The maximum DRAM power consumption is obtained

when $BW_{rd} = BW_{DRAM}$ and $BW_{wr} = 0$. Supposing to have a 4-channel ($N_{MC} = 4$) DRAM chip, $P_{DRAM,max} = 4(10.9 + 0.8 \cdot 522) = 1714mW$.
The DRAM energy consumption is given by:

$$E_{DRAM} = P_{DRAM} T_{EXE} = P_{DRAM} \frac{NOPS}{Th_S} \tag{4.12}$$

where $NOPS$ is the number of operations performed.
The PE and NoC power consumptions are obtained from post-layout simulations under realistic workload (SAIF back-annotation as explained in section 4.1). Figure 4.4 shows different PE's power consumption curves obtained by varying the size of the input buffer M_{IB} which is a design variable (section 4.3).

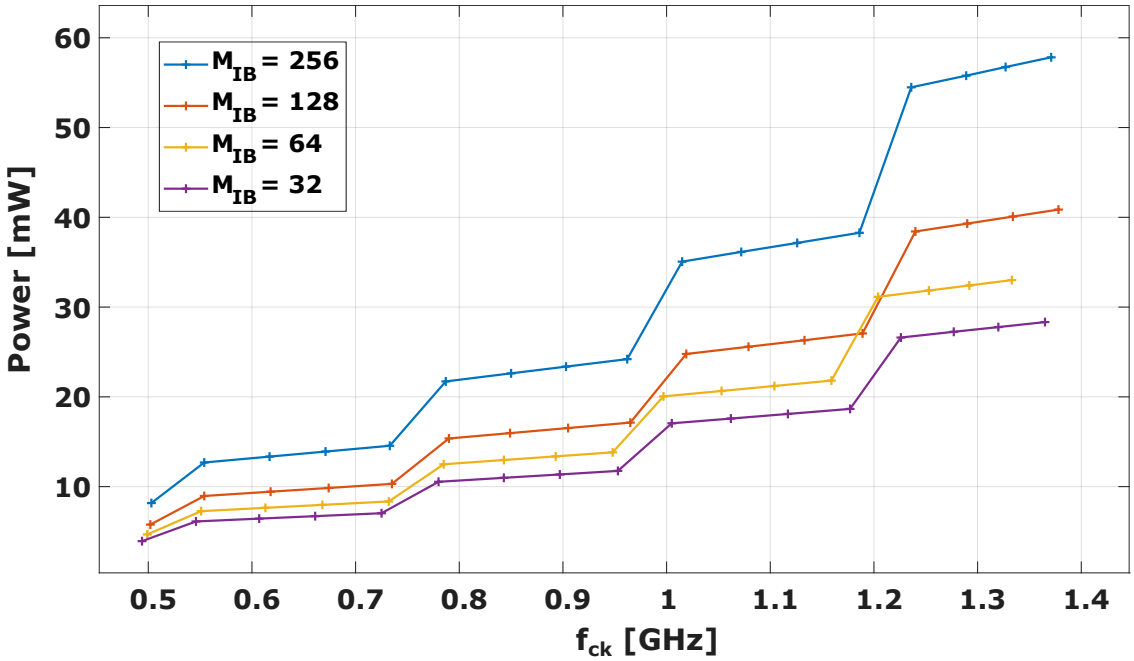

Figure 4.4: PE's power consumption as function of the working frequency. The different curves refer to different values of the size of the input buffer (expressed as number of words).

Even when considering the worst case, which is $M_{IB} = 256$ and $f_{ck} \simeq 1.4$ GHz, the PE power consumption (less than 60 mW) is 30× smaller than $P_{DRAM,max}$.
Finally, the on-chip SRAM energy is obtained from the SRAM module generators. The described model was implemented in Matlab and an extensive design space exploration was carried out, by using the design variables listed in table 4.1. The results of the exploration are reported in the next section.

4.5 Results and Analysis

The solutions obtained by running the exploration are projected into the *throughput-energy efficiency* subspace and then Pareto points are extracted. The obtained Pareto curves are reported in figures 4.5(a)-(b); figure 4.5(a) does not take into consideration the DRAM energy, whereas figure 4.5(b) does. In correspondence of

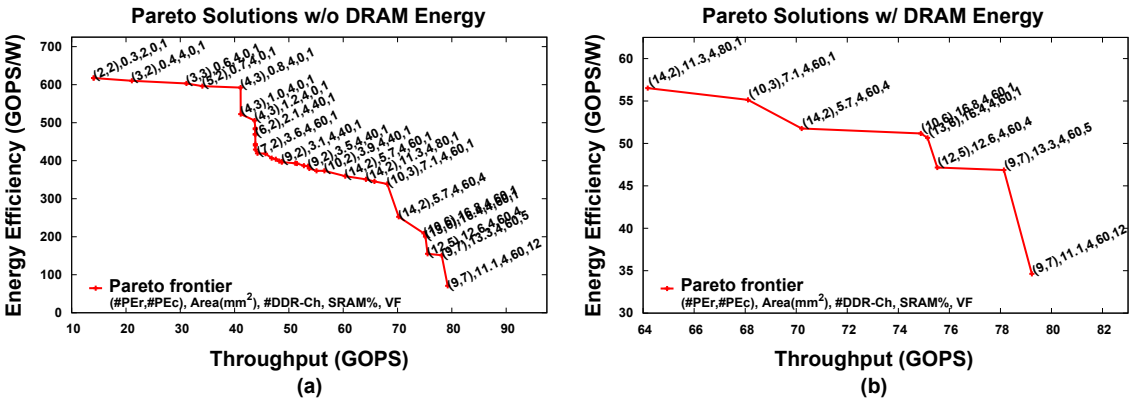

Figure 4.5: Pareto-Optimal Solutions

each Pareto point the most representative design variables, among the ones listed in table 4.1, are indicated. These are: the size of the PE array expressed as pairs of PEs per row N_{PEr} and PEs per column N_{PEc} $(\#PEr, \#PEc)$, the total die are (in mm^2), the number of DDR channels $(\#DDR\text{-}Ch)$, the percentage of total die area dedicated to SRAM $(SRAM\%)$ and the chosen VF pair code $(VF,\ 1 \div 17$ as reported in figure 4.2).

The first and most evident difference between the two plots is the significant impact of DRAM energy on the energy efficiency of the DLP. Indeed, when taking into account the impact of accessing the off-chip DRAM, the energy efficiency drops of one order of magnitude. Furthermore, the Pareto curve is compressed and it is shifted towards higher throughput values. In fact, below 64 GOPS, all the existing implementations show an inferior energy efficiency; by contrast, without the DRAM, highly energy-efficient implementations can be obtained even at a lower throughput, down to 10 GOPS.

A closer look to the Pareto points reveals that the points obtained by considering the DRAM energy are a *subset* of those obtained without the DRAM energy. In particular, only the Pareto points with throughput greater than 64 GOPS are preserved when the DRAM energy is included; the ones that exhibit lower throughput but higher energy efficiency without the DRAM energy are pruned when the DRAM energy is included because they become Pareto-dominated. These pruned points happen to be those with a small area and a reduced parallelism, as reported in figures 4.6(a)-(b). The first Pareto-optimal implementation with DRAM energy (64 GOPS) takes an area ($11.3 mm^2$) that is 37× larger than that of the minimum

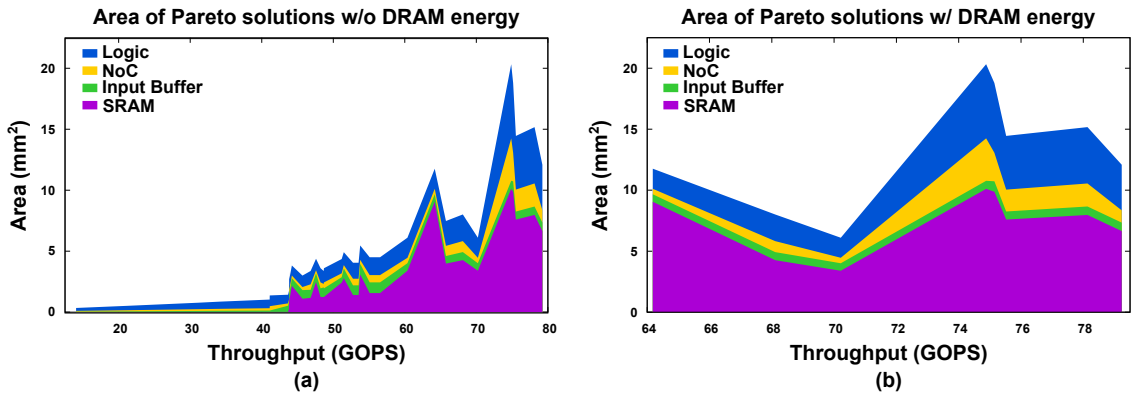

Figure 4.6: Silicon area utilization as function of throughput.

throughput solution obtained without DRAM (10 GOPS). Moreover, one should note that the percentage of SRAM area, which is 0% without DRAM energy for the low-throughput points, dramatically increases when considering higher throughput solutions. This important result can also be observed from a different angle.

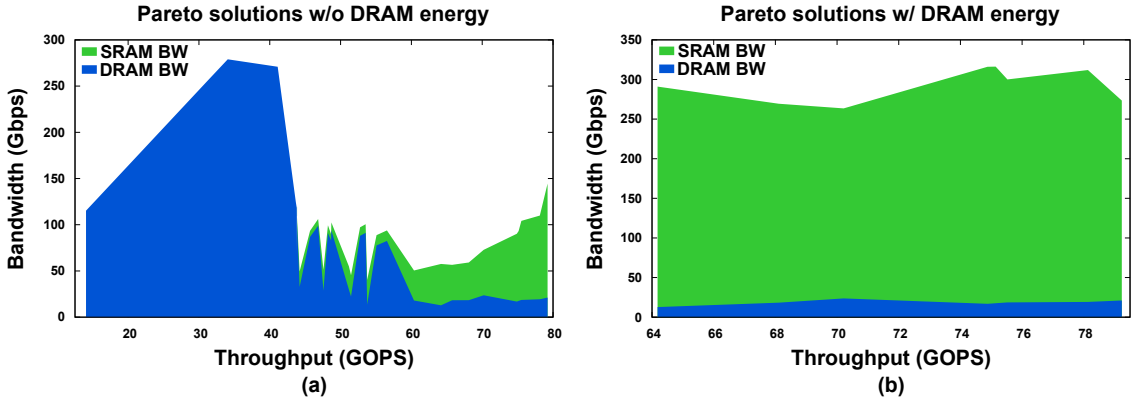

Figure 4.7: Memory bandwidth as function of throughput.

The plots in figures 4.7(a)-(b) show the traffic per unit of time (Gpbs) to/from the on-chip SRAM and the off-chip DRAM. An architectural organization that does not consider the energy cost of moving data off-chip would saturate the DRAM bandwidth first, and then, just for high throughput (>40 GOPS), would start using the SRAM. By contrast, an effective design optimization would keep DRAM bandwidth as low as possible by stressing the SRAM usage; how much SRAM is actually needed can be inferred from the Pareto-analysis of figure 4.5 on the base of the throughput constraint.
For what concerns the VF selection, Pareto solutions use VF pairs numbered from 1 to 12 and, obviously, greater numbers are associated to higher throughput, due to the increasing clock frequency. VF pairs 13 to 17 are only used by dominated

solutions.

Other experiments were carried out in order to determine how the usage of a single-V_{DD} or a dual-V_{DD} DVFS technique affects the space of the solutions. In these experiments the total die area is not a design variable but two boundary cases are considered: (1) $A_{MAX} = 2$ mm^2 and (2) $A_{MAX} = 10$ mm^2. Figure 4.8(a)-(b) depicts the result of the design space exploration when using a dual-V_{DD} DVFS power management (please refer to section 4.2) and $A_{MAX} = 2$ mm^2. Neglecting (figure 4.8(a)) or not the DRAM energy (figure 4.8(b)) causes changes in the space of the solutions that are coherent and analogous to those already explained in the analysis of figure 4.5, i.e. when considering the DRAM energy there is a drastic reduction in terms of energy efficiency, a shrinking of the throughput range and an increase in the percentage of area occupied by the SRAM. In addition to these considerations, one must observe that, when not considering the DRAM energy, highly energy-efficient solutions tend not to maximize the use of the whole available area (2 mm^2) and $SRAM\% = 0$. On the contrary, when the throughput increases, accessing continuously the DRAM would not be energy efficient, hence, SRAM is needed. This results in a higher area occupation. In contrast, when taking into account the DRAM energy, all the Pareto solutions uses the whole available area and allocate some percentage of it for SRAM. For what concerns voltage-frequency pairs, the two cases depicted in figure 4.8(a)-(b) show the same trend: VF pairs increase with the throughput because PEs work at higher clock frequencies, hence, a higher V_{DD} is needed. VF pairs, instead, gets smaller when the energy efficiency increases, confirming the fact that, when approaching the near-threshold region, a high energy efficiency is achieved at lower voltages.

Figure 4.9 shows a comparison between single and dual-V_{DD} DVFS Pareto curves for a maximum target die area of 2 mm^2 and including the DRAM energy. Each point of the Pareto curve is labeled with the code indicating the chosen VF pair. It can be seen that there is an actual advantage in using a dual-V_{DD} DVFS scheme only for high-throughput solutions, while for higher energy-efficient solutions there is no significant benefit. This can be explained by considering that high energy-efficient solutions work at low clock frequencies ($1 \div 4$ VF pairs), thus most (or even all) of the tiles are fixed at low-V_{DD} (no or few critical timing paths, as explained in section 4.2). As a consequence, there is no space for optimization. When considering high-throughput solutions, instead, the target working frequency is higher and applying a dual-V_{DD} scheme can make a difference.

Finally, in figure 4.10, dual-V_{DD} DVFS solutions are also reported for the case $A_{MAX} = 10$ mm^2. Since the same considerations about ignoring or not the DRAM energy hold, only the case that takes it into account is shown. It can be noticed that these solutions allocate a big portion (from 40% to 80%) of the total die area for on-chip SRAM. In fact, allocating more area for PEs would not be an efficient solution since, at a certain point, the throughput will be limited by the memory bandwidth, despite the increase of the number of PEs.

Regarding the other design variables listed in table 4.1, the following trends are common to all the experiments conducted.

- The number of memory controllers N_{MC} tends to be maximized with and without the DRAM energy (there is only one solution in figure 4.5(a) that has $N_{MC} = 2$ and it is the one with lowest throughput).
- The size of the input buffers M_{IB} increases for solutions with a higher energy efficiency, while it gets smaller with larger throughput. This is justified by the fact that those solutions tend to favor a silicon allocation with a higher number of execution units for a given area (larger parallelism) and a larger amount of SRAM buffers to sustain the processing throughput. When fixing the maximum die are to 2 mm^2 and 10 mm^2, IBs are larger for the former and smaller for the latter case as more area is allocated for SRAM.
- All the Pareto solutions have $DB_{IF} = 1$ and $DB_W = 0$. In fact, the increase in throughput that can be obtained by using the double buffering mode on weights is almost negligible and, as an additional drawback, it reduces the IB portion dedicated to input features (IF_{IB}).
- Finally, the area allocated to the NoC gets larger as the throughput increases (figures 4.6(a)-(b)); obviously, this is required in order to avoid communication bottlenecks between the PEs and the on-chip or off-chip memory.

The Energy-Performance design space exploration and the Pareto analysis conducted highlight interesting optimization paths that can serve as point of reference to optimize existing architecture or to design new optimal ones. Even though the model described in section 4.4 refers to the DLP presented in chapter 3, it can be easily generalized (the DLP proposed is also generic and flexible), since it already takes into account the most common metrics used to evaluate DL accelerators, and used as an exploratory tool to study different preliminary solutions and help designers choose the one that most fit the constraints.

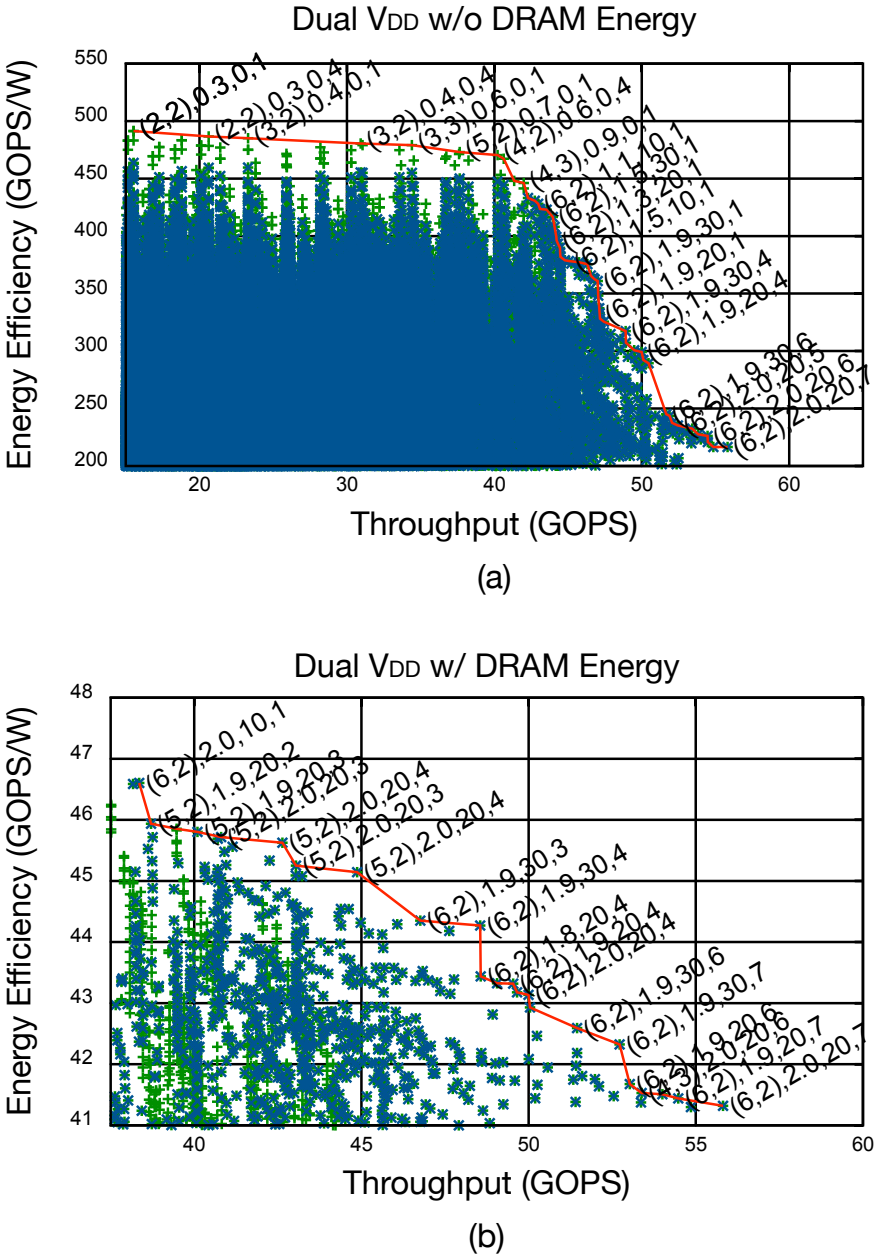

Figure 4.8: Exploration of the dual-V_{DD} DVFS case when $A_{MAX} = 2$ mm^2 and (a) the DRAM energy is neglected, (b) the DRAM energy is considered.

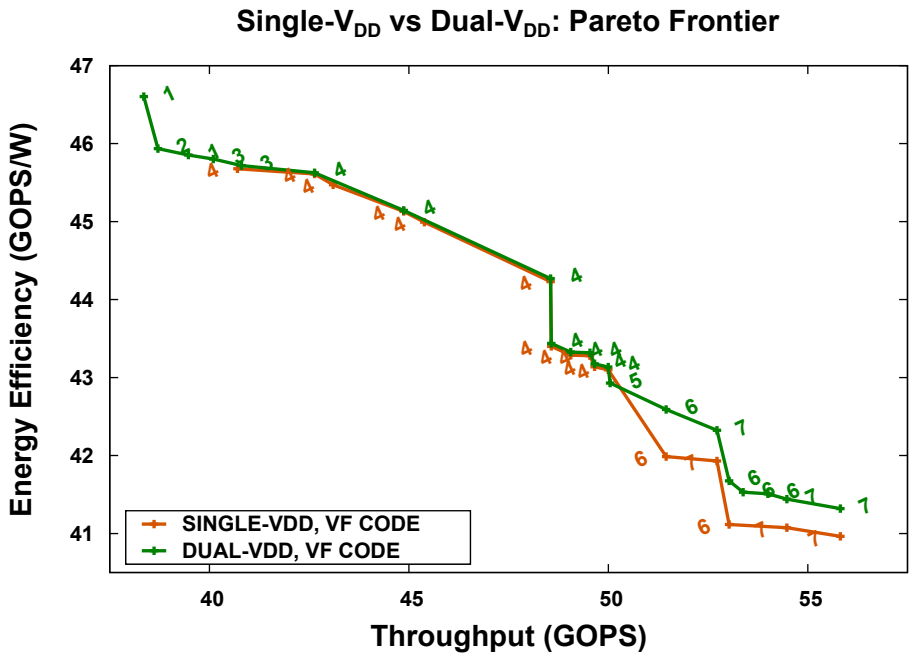

Figure 4.9: Comparison between the Pareto curves obtained when applying single and dual-V_{DD} DVFS. A_{MAX} is fixed to 2 mm^2 and the DRAM energy is included.

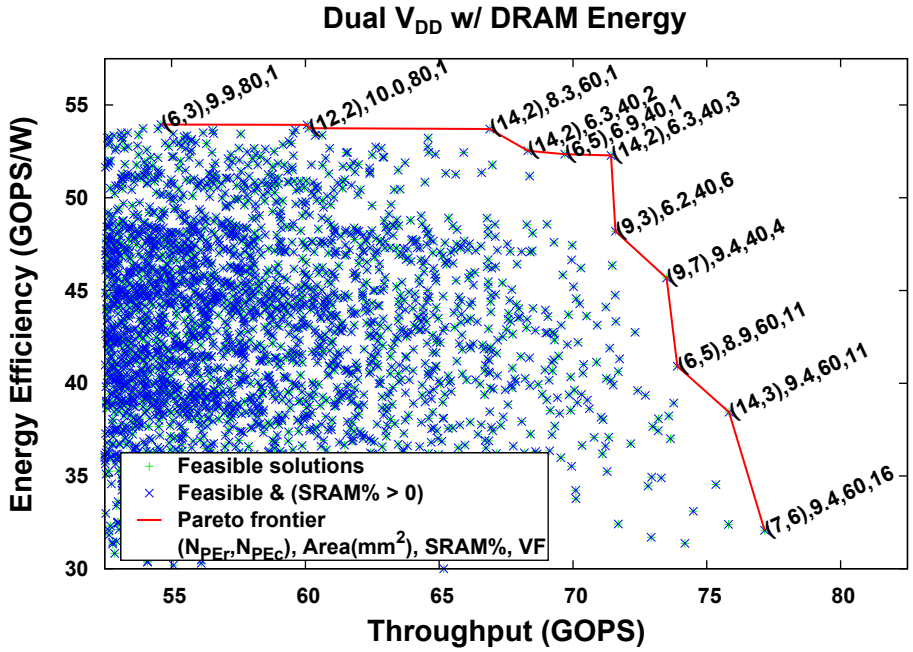

Figure 4.10: Exploration of the dual-V_{DD} DVFS case when $A_{MAX} = 10$ mm^2 and the DRAM energy is taken into account.

Chapter 5

Conclusions

The design of a hardware accelerator for the inference of Convolutional Neural Networks is a challenging task for manifold reasons. First of all, a thorough understanding of Deep Learning and CNNs is fundamental. In particular, when focusing on CNNs one must acquire knowledge of how they are structured, of the different kinds of existing layers and operations involved and of the data flow inside a layer and across multiple layers. Furthermore, it is very important to understand which are the computational and memory bottlenecks that characterize such learning models. Putting together all these information and deriving an optimal architecture is not at all trivial as the number of requirements and constraints is large. In addition, CNN acceleration other than being a hot research topic, it is already widespread in industry as well. As a consequence, giving a novel contribution can be hard.

This work introduces a novel Deep Learning Processor that differs from other solutions for being extremely flexible (as thoroughly discussed in chapter 3) and a sophisticated model (presented in chapter 4) that can be used as a design exploration tool to evaluate different energy-throughput optimal design solutions.

Many optimization paths could be further explored under different aspects: data quantization [86] or compression [45] techniques could be used to reduce the memory footprint and increase the number of data that can be stored on-chip; in addition, data quantization allows to simplify the hardware requirements as it uses a reduced data precision. From a hardware point of view, it could be worth to explore the use of variable-precision execution units as the data precision may vary across the network layers.

Last but not least, it might be effective to migrate towards completely different computational paradigms (beyond the von Neumann model) and explore this field in conjunction with novel emerging technologies. The second part of this thesis will try to address and explore these new approaches.

Part II

Beyond the Von Neumann Paradigm

Chapter 6

Motivations

The term *von Neuman architecture* indicates any computational system composed of a CPU and a memory that stores data used by the CPU. The CPU executes instructions and continuously accesses the memory in order to retrieve the data that it needs. This data exchange between the CPU and the memory is the basis of the von Neumann paradigm [40]. Since it was first proposed in 1945, the technology has undergone huge advances. Thanks to the CMOS technology scaling, transistors are getting smaller and smaller. Nowadays, indeed, it is possible to pack in a single chip billions of transistors. Computational systems are extremely powerful and fast but, in order to sustain the pace, memories should be able to provide as many data as required by the computational core. However, if on one side the CMOS technology keeps making progresses, on the other side memories are not, as the main limitation is represented by the bandwidth. Hence, memories cannot provide the amount of data required by computing units at the same rate of their working frequency. This discrepancy represents the so-called *von Neumann bottleneck* or *memory wall*. Different solutions have been proposed during the years, one of them is memory hierarchy. The principle behind memory hierarchy is to have multiple memories with different sizes placed at a distance from the processing unit that is directly proportional to their size. Therefore the smallest memory is the closest to the processing unit, being also the fastest. The goal of memory hierarchy is to hide the latency of the main memory (the biggest) by adding different intermediate levels of smaller memories. However, this solution is not enough to solve the memory wall problem.
Another critical limitation intrinsic to the von Neumann model is that the continuous data exchange between the processing unit and the memory causes a huge power dissipation. In data intensive applications this problem is exacerbated; indeed, the main part of the total dynamic power consumption is caused by memory accesses.
For these reasons, in the last years, the research and industry communities have focused their attention on alternative solutions that go beyond the von Neumann

paradigm, trying to tackle the problems at the root. Many approaches are being explored: some of them try to bring computation and memory closer, others, such as the *Logic-in-Memory* one, go beyond the separation between computation and memory trying to fully integrate them in a single unit. The advantage brought by the Logic-in-Memory approach is twofold: on one side, integrating logic and memory means exploiting the *full internal memory bandwidth*, on the other, *data are manipulated directly inside the memory* without the need to move them outside for computation and then write back the results.

The von Neumann bottleneck is not the only consequence of CMOS technology scaling. It seems, indeed, that transistor scaling is approaching a boundary not only from the physical point of view but also from the technological and economical one. The well-known *Moore's law* has been obeyed for long and everything possible has been done in order to adhere to it. However, as predicted in the 2013 International Technology Roadmap for Semiconductors [2], 2D scaling will eventually reach some fundamental limits. For what concerns the *physical limitations*, as the device becomes smaller, tunneling and leakages currents increase impacting on the performance of the transistor and on the power consumption. This one is further affected by the number of transistors integrated per unit-area which is in steady growth and that also causes an high thermal dissipation. Regarding the *technological limitations*, lithography-based techniques are not able to provide the required resolution - below the light wavelength - to manufacture CMOS devices. Moreover, the *rising in cost* of production, equipment and testing may reach a point where it will be not affordable from the economic point of view. These are the main reasons why the scientific community is moving towards the introduction of novel *beyond-CMOS technologies* supporting new information-processing paradigms that are potentially able to solve the problems addressed before. Among these emerging technologies, Nano Magnetic Logic is one of the most interesting because it provides non volatility, computing capability and low power consumption. All these key features make Nano Magnetic Logic (NML) a perfect candidate for Logic-in-Memory.

This research work investigates the concept of Logic-in-Memory by presenting a novel Configurable Logic-in-Memory Architecture (CLiMA). An NML-based version of CLiMA is also presented.

Chapter 7

State of the Art

In-memory computation has been extensively targeted in literature and different solutions have been explored. Given the extent of the State of the Art and the differences between the proposed approaches in terms of design and implementation choices, it is very difficult to make comparisons. In this chapter it is presented a taxonomy that categorize the main works present in literature, based on the role that the memory plays in computing data. Four main approaches have been defined and in figure 7.1 the differences between them are highlighted. Moreover, in appendix A it can be found a table that collects the main relevant works found in literature and cited in this chapter or elsewhere in this second part of the thesis. The works are listed in year-of-publishing order and for each of them the following items have been reported: concise description of the architecture, technology used, target applications, software/models used for simulation and/or evaluation, if the proposed architecture is silicon-proven or not, classification according to one of the approaches reported below.
The aim of this categorization is trying to delineate common and divergent features between the works, methods and tools that might be of help – not only for the scope of this thesis but also, and especially, for future works – for comparison purposes and/or for developing new ideas.
It is important to underline that this categorization does not contain all the works that can be found in literature, but the most relevant ones. Nonetheless, as far as it is known, any other work in literature can be identified as belonging to one of the four approaches described below.

7.1 Computing-near-Memory Approach

This approach can be considered as an evolution of conventional architectures where logic and memory are two separated units. In fact, works that belong to this category try to bring closer computation and storage by exploiting *3D Stacked*

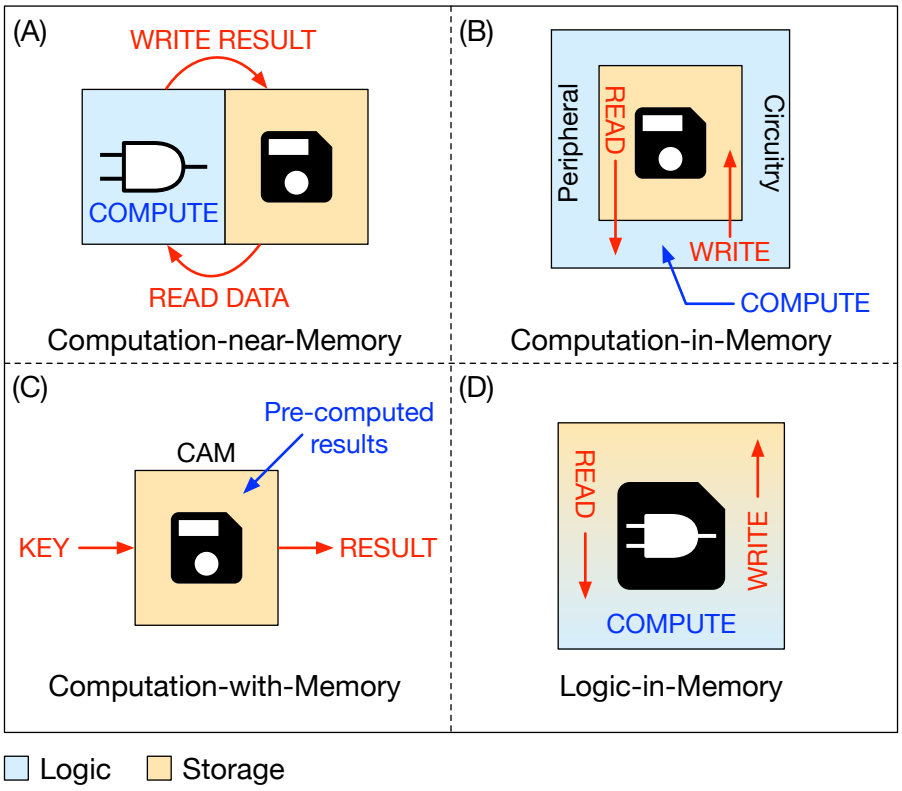

Figure 7.1: Depending on the role that the memory plays in computing data, four main approaches can be defined. (A) Computation-near-Memory: logic and storage are still two separate entities but they are brought closer together thanks to 3D-integration technologies (section 7.1). (B) Computation-in-Memory: memory is used as is to perform computation (section 7.2). The actual data manipulation takes place in the peripheral circuitry (e.g. sense amplifiers) of a memory. (C) Computation-with-Memory: memory is used (as a CAM) for storing pre-computed results for LUT-based computation (section 7.3). (D) Logic-in-Memory: simple logic is added inside the memory cell to manipulate data locally (section 7.4).

Integrated Circuit (3D-SIC) technology [1]. This 3D integration technology allows to stack silicon wafers or dies one on top of the other by interconnecting them vertically, exploiting *through-silicon vias* (TSVs). As the name suggests, a TSV is an electrical connection that crosses a silicon wafer, connecting the two sides of it. By exploiting a 3D-stacking technology, not only memories can be implemented as 3D structures with several memory layers stacked one on top of the other (such as the Hybrid Memory Cube, HMC [55] or the High Bandwidth Memory, HBM [49]), but they can also be stacked on top of a computing unit. Computation and memory are very close but still separate entities. For this reason, this approach has been named *Computation-near-Memory* (CnM). Moreover, as shown in figure

7.1(A), there is data movement from the memory, when data are read and sent to the computing unit, and to the memory, for writing back results.
The advantages of TSV technology are manifold:

- shorter interconnections;
- wider memory bandwidth;
- reduced power consumption;
- more functionality (and/or storage capacity) in a smaller area.

All these benefits help mitigate the memory-bottleneck problem.
Some relevant works are [66], [125], [3], [124], [111], [119]. In particular, in [66], [125] and [3] authors propose a multi-core architecture where each processor is composed of two tiers, a computing one and a memory one stacked on top of it. In [119] and [124] a host processor (a GPU in the former paper and a CPU in the latter, non 3D-stacked in both cases) delegates data-intensive tasks to 3D-stacked computation-near-memory units that act as co-processors. Authors in [111] and [119] exploit the logic layer at the base of the HMC to perform near-memory computing. Works such as [31] and [107] do not make use of a 3D-stacking technology, but they propose to integrate the processor and memory banks on the same silicon die to exploit a direct connection between them and avoid off-chip communication. The benefits of such architectures are demonstrated by means of memory-intensive and parallel benchmarks. Moreover, all the works cited here are implemented in CMOS technology.

7.2 Computing-in-Memory Approach

Works belonging to these category use non-volatile memory technologies, such as MRAM (Magnetoresistive Random-Access Memory) [7] or RRAM (Resistive Random-Access Memory) [123], or volatile memory technologies, like DRAM (Dynamic Random-Access Memory) or SRAM (Static Random-Access Memory), to perform both computation and storage tasks, by using technology *as is*. This means that the memory array is not modified in terms of structure and functionality, but its analog functionality is exploited to perform operations inside the array. To be more specific, peripheral circuitry (i.e. sense amplifiers, SAs) is modified and used to perform row-wise or column-wise logic operations. Hence, in-memory computation is performed by reading data which is sensed and processed by SAs and the result is written back in the array, as depicted in figure 7.1(B). In this case, data is not moved out of the memory for the computing task, hence the name *Computing-in-Memory* (CiM).
Works such as [6], [23], [44], [37], [76], [82] use a RRAM array to perform logical

or arithmetic operations depending on the target application. Likewise, authors in [101] propose to modify a commodity DRAM to perform bulk bitwise logic operations inside the memory. In [57] RRAM arrays are connected by means of configurable interconnects to realize fast in-memory adder trees. The configurable interconnects can also be used to shift data.
In [14] and [84] authors propose a hybrid architecture in which CMOS logic is used to perform instruction fetch and decode or other functions needed to manage the data computation done inside the RRAM array. In [109] a chip for energy-harvesting applications is presented. The system is composed of a host CPU and RRAM arrays used as co-processor to accelerate ANN-like computing.

7.3 Computing-with-Memory Approach

As the previous one, this approach is also based on performing computation and storage tasks by using non-volatile memory technologies. However, in this case, RRAM arrays are used as Content Addressable Memories (CAM) to perform computations in form of look-up tables (LUT). In fact, any n-input boolean function can be encoded in an n-bit LUT by storing the truth table of the function in the LUT. For example, the addition between two n-bit values can be implemented by storing in the LUT all 2^{2n} input combinations. Pre-computed results are instead stored in a further memory. Then, the LUT is accessed, like a CAM, by using an input key (the combination of the two inputs) and an address is retrieved. This address is used to access the memory that stores the correspondent result of the operation. As shown in figure 7.1(C), computation is carried out with the memory to retrieve a pre-processed result, from which the name *Computing-with-Memory* (CwM).
Works belonging to these class are [53], [65], [121], [58], [59], [85]. In the last cited work, authors propose to integrate RRAM CAM arrays inside the Floating Point Units of a GPU, to hold highly frequent pre-computed values.

7.4 Logic-in-Memory Approach

Finally, in this approach simple logic is directly integrated inside the memory cell, hence the name *Logic-in-Memory* (LiM). Differently from all other approaches, this one enables data computation to be performed locally without the need to move data to the peripheral circuitry or outside the memory. As highlighted in figure 7.1(D), data movement is exclusively internal to the memory array. Readings are done to move data from one cell to another for computation purposes and writings are done to update a cell content after a processing task. The are few works in literature that are part of this class. In [79] authors propose a hybrid memory cell in which an MTJ (Magnetic Tunnel Junction) device, used as non-volatile storage,

is integrated with simple CMOS logic to realize a non-volatile logic-in-memory cell. MTJs are at the base of MRAMs and they are composed of two ferromagnets separated by a thin insulator layer [64]. The resistance of the MTJ changes depending on the relative orientation of the magnetization of the two ferromagnetic layers. In particular, in a parallel magnetization configuration the resulting resistance of the MTJ is low, while in an antiparallel magnetization configuration the resulting resistance of the device is high. The switching between these two configurations can be used to write a logic 0 or 1 in the MTJ [56]. The cell proposed in [79] is exploited in [60] where authors present an architecture based on an hybrid MTJ/CMOS CAM engine for search operations. In this case, the single non-volatile logic-in-memory cell is a 3D structure in which a MTJ cell is stacked on top of simple CMOS logic. In [120] authors propose a modified SRAM array in which rows of logic cells (LUT-based and XOR-based) and rows of memory cells are alternated according to a memory-logic-memory-latch scheme. The latch rows are used as redundant storage cells to hold partial or final results.
The system presented in [62] is a hybrid approach between Computing-near-Memory and Logic-in-Memory. The proposed architecture exploits 3D integration by stacking DRAM on top of a logic layer using TSVs and it also adds logic inside the DRAM to perform XOR operations. More specifically, while in the other works belonging to this class logic is added directly inside each memory cell, in this work XOR engines are added outside each memory bank.

Chapter 8

CLiMA: Configurable Logic-in-Memory Architecture

This chapter presents a novel approach to the concept of Logic-in-Memory, that has brought to the definition of a new architecture called *CLiMA*, Configurable Logic-in-Memory Architecture. The main idea behind CLiMA is the definition of a *flexible architecture* that can adapt to different applications and requirements. In fact, as the name suggests, CLiMA is a configurable architecture, since it can be configured to perform different types of data computation, that exploits the approach of Logic-in-Memory to enable data computation in memory. However, CLiMA does not rely exclusively on the LiM approach, but, if necessary, it can integrate one or more of the other approaches presented in chapter 7. In this sense, the configurability of CLiMA extends also to the concept of in-memory computation.

8.1 Overview

As shown in figure 8.1, CLiMA can be seen, in its most generic form, as an heterogeneous system that exploits different degrees of in-memory computing to reduce as much as possible the memory bottleneck problem and its related consequences, as thoroughly explained in chapter 6. As explained before, the idea is not to limit CLiMA to exploit a single approach but a combination of them, if necessary. The reason behind this choice is that, depending on the target application's characteristics, there might be a part of the algorithm that well maps, for example, on a Logic-in-Memory scheme, while other parts do not. Hence, operations that are suitable to be implemented in-memory (e.g. logic operations) are executed by CLiM arrays, while more complex operations (e.g. multiplication, division) are executed by a dedicated logic that cannot be implemented directly in memory. In figure 8.1 the dedicated logic with its own memory represents the CnM unit of CLiMA.

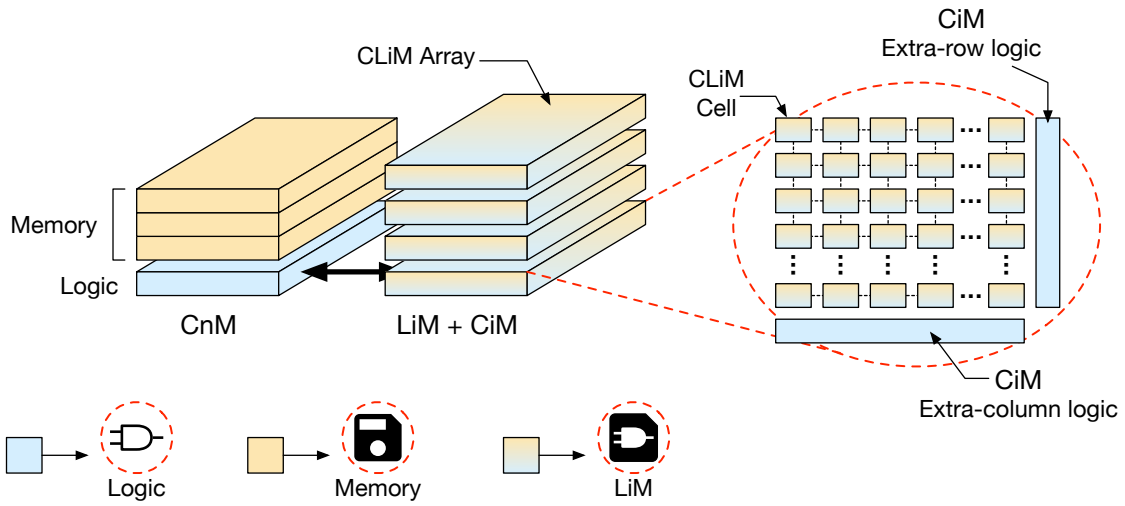

Figure 8.1: CLiMA as an heterogeneous system where different approaches (CnM, LiM, CiM, CwM) are integrated together in order to guarantee maximum flexibility.

CLiM arrays represent, instead, the LiM unit. Internally, a CLiM array has several configurable cells that integrate logic and storage (CLiM cells). CLiM cells can be interconnected in different ways, depending on the data exchange required by the target algorithm. Moreover, for some applications there might be the need for further data processing outside the rows or the columns of the array, which is the aim of the extra-row/column logic. This logic can be considered as the CiM unit of CLiMA. Even if not represented in figure 8.1, CLiMA could even have a CwM unit (i.e. a CAM memory) for LUT-like computation.
It is clear that the flexibility of CLiMA is twofold:

1. provide support for different applications and for different types of operations (logic and arithmetic);

2. provide different degrees of in-memory computation.

Figure 8.2 shows a high-level block diagram of CLiM array. The green and red boxes indicate a row and a column of the array, respectively. Data manipulation can happen locally, inside each LiM cell, or between cells. Moreover, data can be manipulated externally to rows and columns by the extra-row/column logic. Figure 8.3 depicts, more clearly, the different possible types of data manipulation that can take place inside the array. As before, green and red boxes indicate rows and columns, respectively. Five possible in-memory types of operations can be defined:

- Local: data is manipulated locally, inside the cell;

- Intra-row: an operation takes place between two or more cells inside the same row (black dashed arrow in figure 8.3);

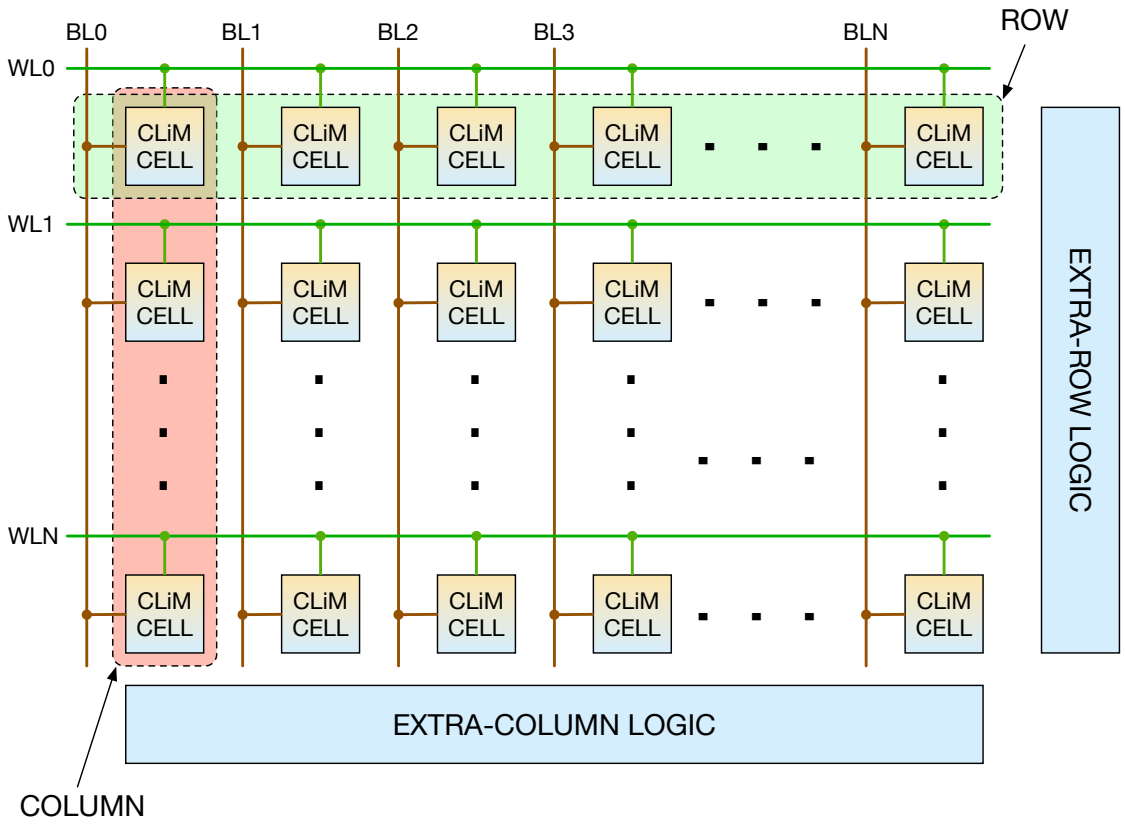

Figure 8.2: High-level block diagram of CLiM array.

- Intra-column: an operation takes place between two or more cells inside the same column (black solid arrow in figure 8.3);
- Inter-row: an operation that involves two rows, as instance an operation between a data stored in row A and one stored in row B;
- Inter-column: an operation that involves two columns, as instance an operation between a data stored in column A and one stored in column B.

Inter-row/column computations are perfectly fit for bitwise operations between two rows or columns. Intra-row/column operations can, instead, be used to build more complex data-flows to enable complex in-memory computations. An example is depicted in figure 8.34. Each CLiM cell is represented as a logic-enhanced memory cell. The logic in each cell is composed of a configurable logic block that can be configured to perform boolean logic functions (e.g. AND/OR/XOR) and a full adder to perform additions. By exploiting intra-row operations, each memory row can work as a Ripple Carry Adder (RCA), highlighted by the red box in each row. By exploiting intra-column operations (in addition to inter-row operations), more

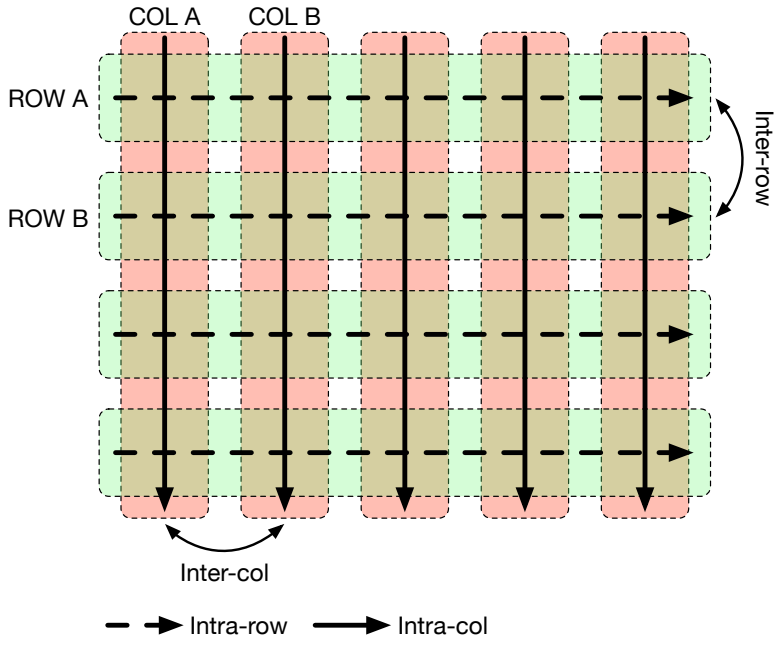

Figure 8.3: Possible types of data manipulation inside CLiM array.

memory rows can work as an Array Multiplier (AM). In fact, RCAs inside each row produce partial products, then, these partial products are accumulated vertically to produce the final multiplication result. Various multiplexers are used to direct signals coming from/going to other cells. Figure 8.4 shows a magnified view of the structure depicted in figure 8.34. The memory block (MEM) of each CLiM cell is accessed by using word-lines (WL) and bit-lines (BL). Each CLiM cell can be used to perform a logic operation or a sum between the data stored locally (in MEM) and another data that can either be an external input (EXT_IN) or a data coming from another cell. Looking at signals ① to ⑤ in figure 8.4, it is possible to distinguish among the following connections:

- Intra-cell connection

  ① write back the result from the logic (L) or full adder (S) into MEM.

- Inter-cell connections

  ② Cell output to south cell used for:
  - inter-row operations;
  - intra-column operations.

  ③ Cell output to east cell used for:
  - inter-column operations;

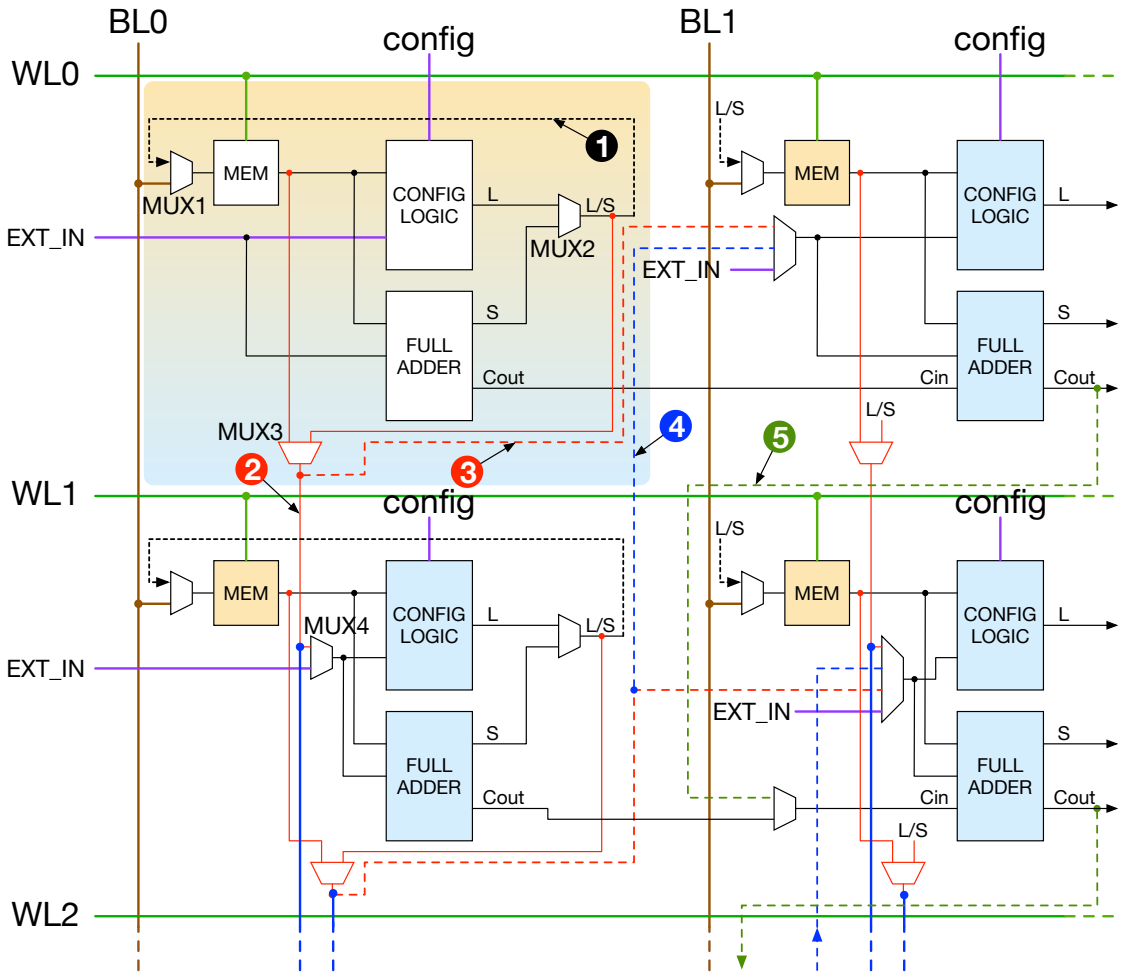

Figure 8.4: Detail of connections inside and between cells in a CLiM array. Some connections (④ and ⑤) are specifically designed to support in-memory RCA and AM.

  - intra-row operations.

④ Cell output to north-east cell used for:

  - in-memory AM support;
  - diagonal data movement (inter-row and inter-column movement).

⑤ Output carry to south cell used for:

  - in-memory AM support.

MUX1 is used to select which data to write inside the MEM block: if the external data coming from the bit-line (to initialize the content of the memory) or the data computed locally (the content of the memory is updated). MUX2 is used to choose

among the result coming from the logic block (L) or the one coming from the full adder (S), depending on the type of operation needed. MUX3 allows to choose which data to send to neighboring cells: if the one coming from logic/full adder or the one saved locally. This choice depends on the data-flow of the target application and on how it is mapped on CLiMA. Finally, MUX4 is used to choose the source of one of the two inputs of the logic block or the full adder. The number of inputs of MUX4 changes depending on the position of the cell inside the array. As instance, the very first cell on the first row and first column has only one input, hence the absence of MUX4. The other cells, instead, have two or more inputs depending on how many connections are designed to interconnect them.
It is clear that the more the connections the more the possibility of building complex in-memory functions and data-flows. To sum up, the architecture showed in figure 8.34 can support several types of in-memory operations and data movement between cells:

- in-memory logic operations;
- in-memory arithmetic operations;
- horizontal (intra-row/inter-column), vertical (intra-column/inter-row) and diagonal data movement (a mix of inter-row and inter-column).

It is important to highlight that while support for in-memory addition and multiplication is a huge advantage because data are manipulated directly inside the memory, a RCA and an AM are not the fastest arithmetic circuits, hence, performing these operations in memory could slow down the execution. A solution to this problem could be, as instance, to delegate the multiplication (an AM is slower than a RCA) to a dedicated and faster unit outside the memory. This choice would also simplify the interconnection structure inside the array, as some of the connections depicted in figure 8.4 could be eliminated. However, these design choices need careful consideration as some applications might benefit from them and some other might not. If the aim is to work toward the implementation of a configurable and flexible structure that can support different work-loads, data-flows and operations, exploring various applications is the way to understand how the requirements, in terms of hardware resources, functionality, level of in-memory computation, degree of flexibility, vary accordingly. For this reason, some suitable algorithms have been identified and they will be presented in the next section. Moreover, different CLiMA approaches have been explored. The main one is presented in this chapter at section 8.3, while others will be referenced in chapter 9.

8.2 Algorithms Selection

In this section, the algorithms selected for exploring different CLiMA approaches are presented. As a key requirement, the target applications must be characterized

by some common features, listed below.

- *High data demand*: memory intensive applications take advantage from in-memory computation as it allows to reduce or, in same cases, avoid at all data movement from/to the memory as it happens in Von Neumann-based systems.
- *High level of parallelism*: computationally intensive applications that can be executed in parallel to improve performance, benefit from CLiMA as it is, intrinsically, a massively parallel architecture.
- *Simple operations*: in order to exploit in-memory computation as much as possible, applications should be characterized by simple operations (mainly logic operations but, as it will be shown in section 8.3, also some more complex operations can be integrated inside the memory array).

By following these guidelines, several applications have been chosen, ranging from query processing in databases to encryption, neural networks and decision trees.

8.2.1 Database Search using Bitmap Indexes

Being able to retrieve a data from a database at high speed is fundamental for improving performance of database search operations. For this purpose, data structures such as database indexes are used to locate data in a database at a fast rate. There exist different types of indexes. Among these, *bitmap indexes* [28] provide a way to answer database queries only by performing bitwise logic operations.
An example will help explaining better how database search with bitmap indexing works. Let suppose to have a database similar to the one represented in table 8.1. It can be seen that, in this example, gender can assume only two values (F for

Employee ID	Gender	Age range
10	F	A
11	M	B
12	F	B
13	F	C
14	F	D
15	M	A
16	F	E

Table 8.1: Portion of database containing a company's employees information.

female and M for male) while the age range can assume five different values (letters

indicates different age ranges, as instance, A include people that are 25 to 30 years old). Each column in the database can be transformed in bitmap indexes, i.e. vectors of bits. As instance, the bitmap index for the gender column is the one in table 8.2. Basically, the bitmap index of a column will be a set of bits vectors, one

Employee ID	**Gender = F**	**Gender = M**
10	1	0
11	0	1
12	1	0
13	1	0
14	1	0
15	0	1
16	1	0

Table 8.2: Bitmap index of the gender column shown in table 8.1.

for each value that the column can assume. Hence, the bitmap index of the gender feature has two vectors of bits. Answering a query such as 'How many employees are female and their age range is A or B?' can be performed by executing logic operations on the bitmap indexes, as depicted in figure 8.5. The result of the query

Figure 8.5: Query processing using bitmap indexes. Taking as reference the database shown in table 8.1, answering to the query 'How many employees are female and their age range is A or B?' translates into two simple bitwise logic operations. First, a bitwise OR between the bitmap indexs related to Age Range A and B. Then, a bitwise AND between he result of this first operation and the bitmap index related to Gender = F.

can be reused to perform other operations to answer more complex queries.

8.2.2 Random Decision Forests

Random decision forests [50] are a learning method used for classification. A

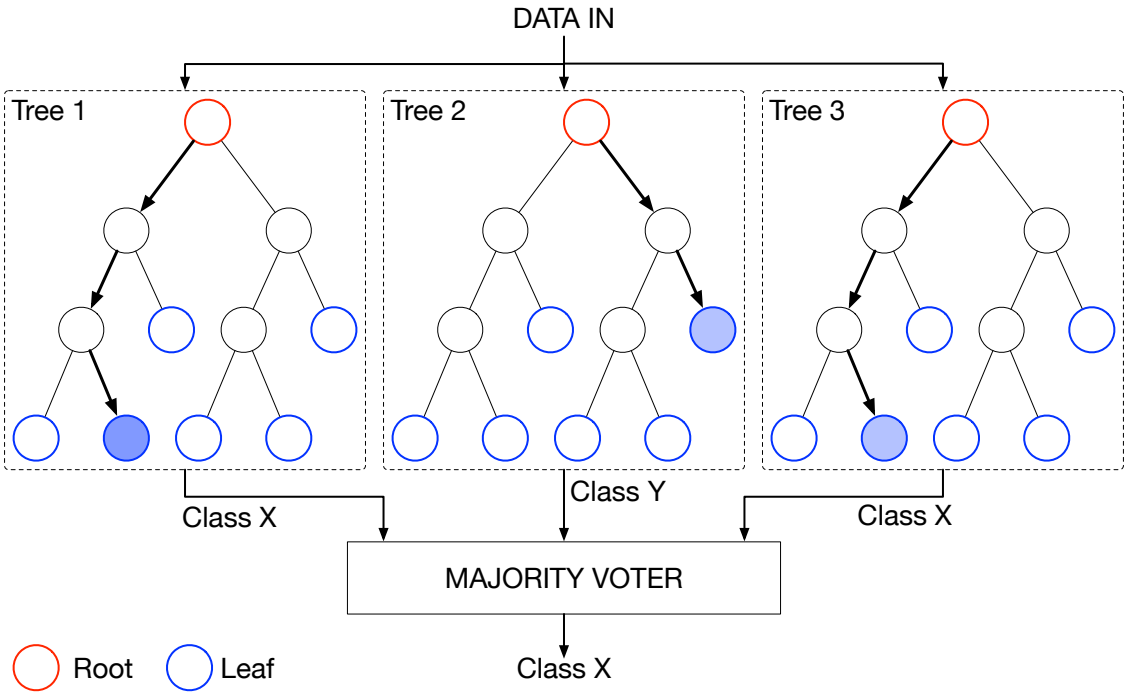

Figure 8.6: Random decision forest.

random forest is composed of a number of decision trees (figure 8.6), all executing the classification task in parallel in order to maximize the classification accuracy. Each tree in the forest is trained on a different subset of the training set. The final classification is computed as majority function of the classification results given by each tree within the forest. Trees are characterized by a certain number of nodes, the first one being the root node and the last ones being the leaf nodes. In order to obtain a classification result, the tree must be traversed starting from the root until a leaf is reached. At each node, a comparison between the input data and a threshold associated to the node is performed: depending on the result of the comparison, the next active node will be the one on the left or on the right. The comparison is then repeated until a leaf node is found, which returns the classification result.

8.2.3 Advanced Encryption Standard (AES)

The Advanced Encryption Standard (AES) [26][32] is a cryptographic algorithm developed by Joan Daemen e Vincent Rijmen. This algorithm can be used to encrypt and decrypt information in blocks of 128 bits, by using a cryptographic key that can have three different lengths: 128, 192 or 256 bits. The 128-bit blocks are organized in 4×4 arrays of bytes, called states. These arrays are given as input to the AES cipher which applies several transformation rounds on the data, called

plaintext, producing the encrypted output data, called cyphertext. The number of rounds depends on the length of the cipher key:

- 128-bit key = ten rounds;
- 192-bit key = twelve rounds;
- 256-bit key = fourteen rounds.

Each round consists of several processing steps described below.

1. *SubBytes*: each byte in the state array is substituted with another byte retrieved from a look-up table.
2. *ShiftRows*: rows of the state array are shifted to the left by a number of positions that varies for each row (the first row is not shifted at all).
3. *MixColumns*: each column of the state array is transformed by applying an invertible linear transformation. The transformation consists of performing very simple operations such as shift and XOR.
4. *AddRoundKey*: each byte of the state array is processed by using a subkey derived, for each round, from the main key. This step consists in performing XOR operations between the bytes of the state array and the bytes of the subkey.

These four steps are repeated for each round, except for the last one in which the MixColumns step is not executed. Moreover, before the rounds are executed, other two preliminary steps are performed:

1. *KeyExpansion*: each round uses a different key that is derived from the cipher one; in this step round keys are computed.
2. *AddRoundKey*: each byte of the state array is transformed by combining it with a portion of the round key using XOR operations.

8.2.4 Quantized Convolutional Neural Networks

Convolutional Neural Networks have been extensively explained in chapter 1. Here, two quantized versions of classical CNNs are presented. In both cases computation and memory requirements are greatly simplified at the expenses of a small loss in prediction accuracy.

XNOR-Networks XNOR-Networks [87] are CNNs in which weights and input activations to convolutional layers are both binary. This means that values are represented on a single bit and they can be equal to +1 or -1, where +1 corresponds to logic 1 and -1 to logic 0. As a result, the standard convolution operation can be approximated with XNOR and bitcount operations.

$$\mathbf{I} * \mathbf{W} \approx (\text{sign}(\mathbf{I}) \circledast \text{sign}(\mathbf{W})) \odot \mathbf{K}\alpha \tag{8.1}$$

In equation 8.1 $\mathbf{I}$ is the input activation matrix, $\mathbf{W}$ is the weight matrix, $*$ is the convolution operator, $\circledast$ is the binary convolution (i.e. XNOR and bitcount), $\odot$ is the element-wise product and $\mathbf{K}\alpha$ is a scaling factor. By doing so, the number of non-binary operations is greatly reduced with respect to the binary ones. This approximation not only simplifies the hardware requirements of such kind of networks, but it also allows to obtain considerable memory savings since inputs pixels and weights require a single-bit data representation.

ShiftCNN ShiftCNN [43] uses a power-of-two weight representation that eliminates the need for multiplications in Convolutional Neural Networks. In this case multiplication operations reduces to simple shift operations. In particular, since all weights are quantized to power-of-two values of the type 2^{-n}, all shift operations are arithmetic right shifts.

8.3 CLiMA for Quantized Convolutional Neural Networks

Following the ideas and concepts presented in section 8.1, here, it is proposed a version of CLiMA for quantized Convolutional Neural Networks. In particular, the reference network in this case is ShiftCNN. An overview of the proposed architecture of CLiMA is given in figure 8.7. The Logic-in-Memory core of CLiMA is represented by the CLiM Array (highlighted by the red box). The array is composed of a number of rows and columns of CLiM cells, each having computation (configurable) and storage capabilities. In this scheme each CLiM cell is intended to be an N-bit cell. Surrounding the array there are row and column decoders used to enable cells for reading/writing operations and also to control the data movement inside the array when performing computations. These decoders are modified in order to enable more contiguous rows and columns at the same time. Row and column masks are used to enable/disable rows and columns in a more fine-grained way. The output decoder is used to select one output data among all the data stored in the array. External to the array there is also the weight memory, a standard memory where convolution weights are stored. The weight dispatcher is used to properly distribute weights among the rows of the array.

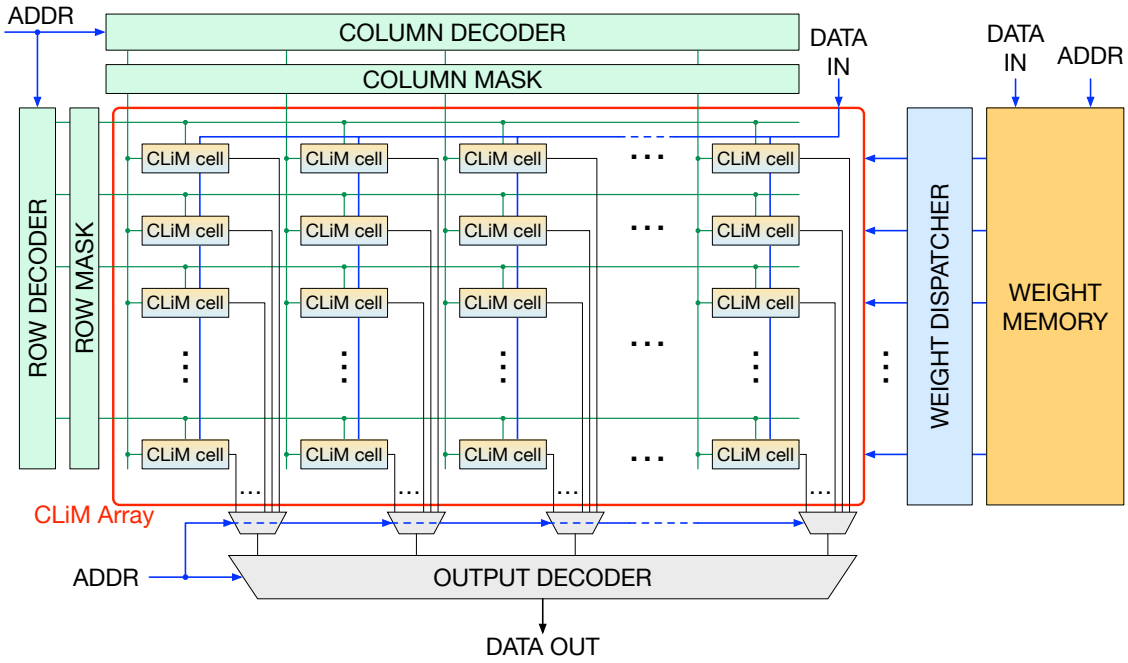

Figure 8.7: Overview of the proposed architecture of CLiMA.

A thorough description of the architecture of CLiMA for quantized CNNs and its usage to perform in-memory computations will be given in the following.

8.3.1 CNN Data Flow Mapping on CLiMA

As explained in section 1.3, the real workload of CNNs is represented by convolutional layers. The computation in such layers consists in shifting the weight kernel all over the input feature map, performing a weighted sum of the inputs, as image 8.8 depicts. It can be observed that convolution windows partially overlap. In order to allow parallel computation of convolution windows, what is usually done is mapping the convolution into a matrix multiplication, as shown in figure 8.9. When moving from a standard convolution flow (figure 8.9(A)) to a matrix multiplication one (figure 8.9(B)), it is as convolution windows were unrolled. When unrolling, as it can be noticed from figure 8.10, weights are shared across pixels on the same column.Hence, multiplications are executed in the vertical directions and accumulations in the horizontal direction. However, the drawback of unrolling convolution windows is the introduction of *data redundancy*. In figure 8.9(B) input data highlighted in red represent the redundant data. These are the input features that belong to the overlapping regions between convolution windows. The number of redundant data clearly depends on the dimensions of the overlapping regions which, in turn, depend on the kernel size and on the stride. In particular, the

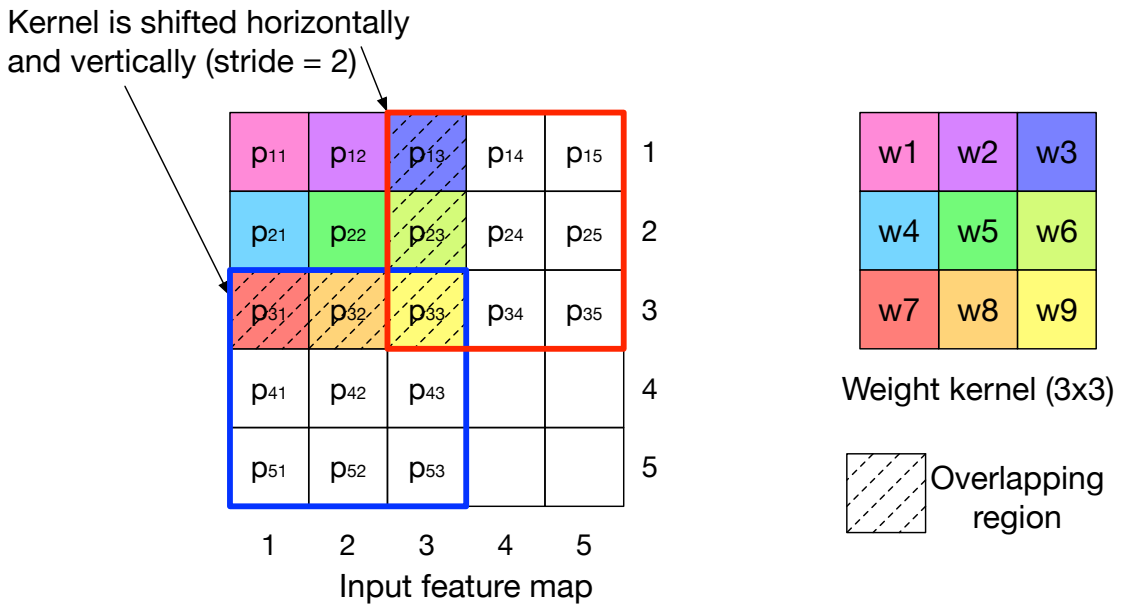

Figure 8.8: The weight kernel is applied on sub-regions (convolution windows) of the input feature map in order to perform convolution. When the kernel is shifted from one sub-region to the other, there is an area of overlap between convolution windows whose dimension depend on the kernel size and stride.

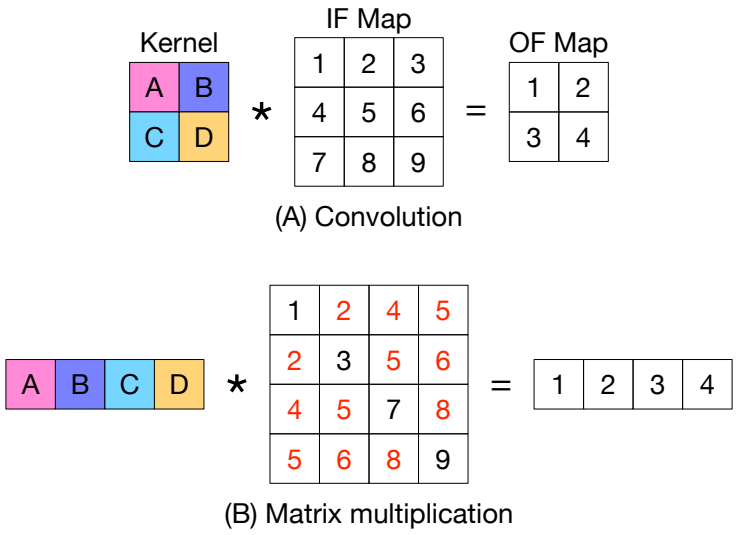

Figure 8.9: (A) Standard convolution. (B) Matrix-multiplication-like convolution. Red numbers indicate data redundancy. IF Map stands for Input Feature Map, OF Map for Output Feature Map.

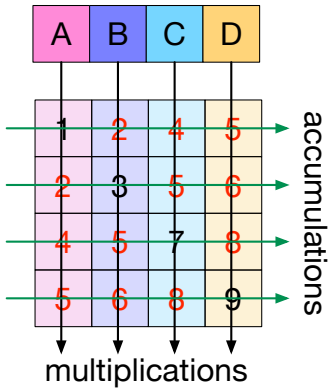

Figure 8.10: Column-wise weight sharing when the convolution window is unrolled.

overlapping region gets larger when the kernel gets bigger and/or the stride gets smaller, as highlighted in figure 8.11.

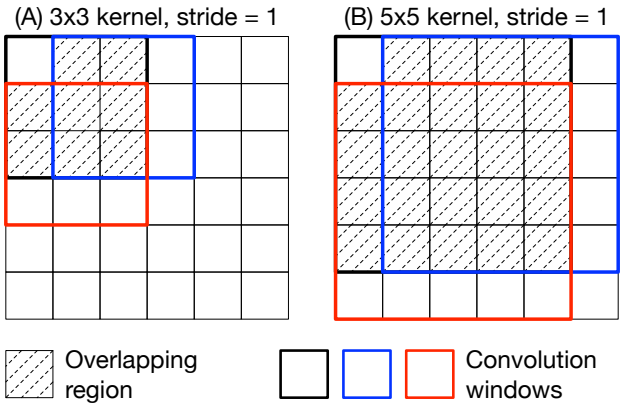

Figure 8.11: Size of overlapping region gets larger when (A) the stride gets smaller and (B) the kernel is bigger.

AlexNet has been taken in consideration as a case study in order to assess how much the number of input features increases because of the data redundancy introduced by the unrolling of convolution windows in a real CNN. The histogram reported in figure 8.12 shows how the number of input features per convolution layer veries before and after the unrolling of convolution windows. The five convolution layers of AlexNet are displayed on the x-axis, while input features are reported on the y-axis (in base 10 logarithmic scale). It can be seen that when unrolling, the number of input features increases of one order of magnitude with respect to the standard situation. This is valid for each convolution layer.
The data redundancy caused by unrolling is not negligible and, an architecture such as CLiMA would not benefit from such unrolled data flow. Indeed, the memory array would not be exploited in an efficient way. On the other hand, by exploiting

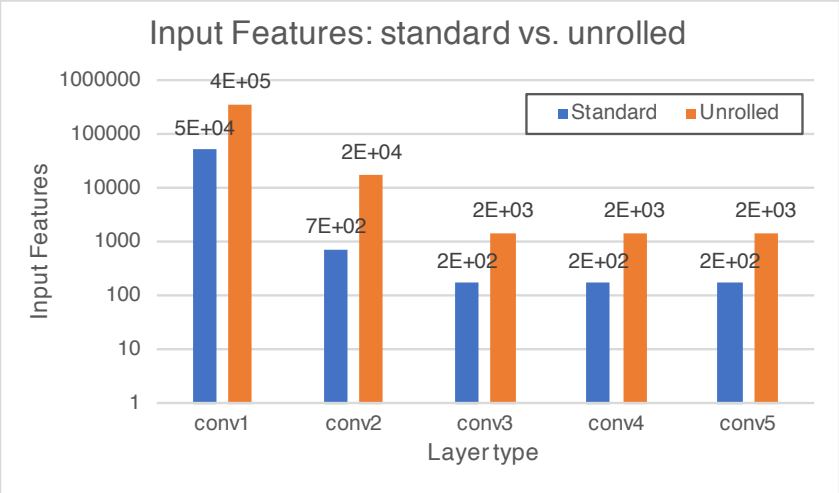

Figure 8.12: Number of input features per convolution layer before and after the unrolling of convolution windows in AlexNet.

unrolling, as shown in figure 8.10, all convolution windows can be executed in parallel inside CLiMA and the convolution computation is straightforward. In fact, supposing that each cell inside the CLiM array stores a pixel, multiplications can be done by distributing weights in the vertical direction (same weights for cells on the same column), while accumulations can be executed inside each row by exploiting inter-cells connections.
However, as already highlighted, the data redundancy caused by unrolling is not acceptable for an architecture such as CLiMA, since the storage space must be used in the most efficient way possible. Therefore, in order to avoid data redundancy and guarantee parallel computation at the same time, a different data flow mapping scheme has been studied. When considering the convolution of a kernel over an input feature map, it can be observed that not all convolution windows overlap. Hence, the idea is to execute in parallel only the non overlapping convolution windows and repeat the process until all convolution windows have been executed. This is graphically explained in figure 8.13. In this example the kernel used is 3×3 and the stride is equal to 2. The convolution of the kernel over the input feature map is decomposed in four different iteration steps. In each of these steps non overlapping convolution windows are executed in parallel. The number of iteration steps required to complete a convolution according to this scheme depends on the following parameters:

- size of the input feature map;
- size of the weight kernel;
- stride.

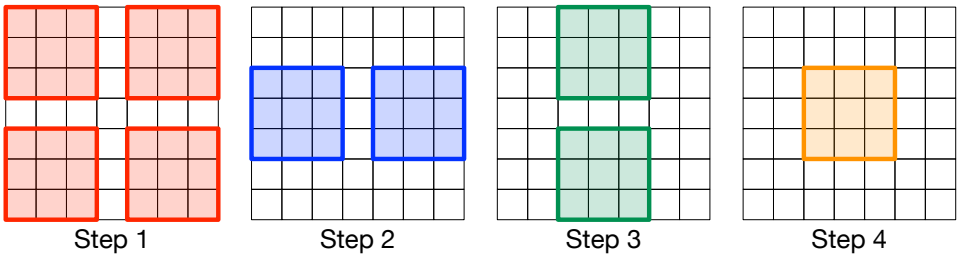

Figure 8.13: Parallel computation of non overlapping convolution windows. In this example it takes four iteration steps to execute the convolution with a 3×3 kernel and stride equal to 2.

This parameters also change layer by layer.
Considering an input feature map with dimensions $W_{in} = H_{in}$, a kernel with size $k \times k$ and stride S, the number of iterations steps can be calculated as:

$$\#iter = \frac{tot\_conv\_windows}{parallel\_conv\_windows} \tag{8.2}$$

where $tot\_conv\_windows$ is the total number of convolution windows, whereas $parallel\_conv\_windows$ is the number of convolution windows that can be executed in parallel. The total number of convolution windows can be simply calculated as $W_{out} \cdot H_{out}$, where $W_{out} = H_{out}$ are the dimensions of the output feature map (as explained in chapter 1, W_{out} and H_{out} are computed according to equation 1.13). The number of parallel convolution windows can be evaluated according to the following equation:

$$parallel\_conv\_windows = \left(\frac{W_{in}}{k + (S-1)} \right)^2, \; k > 1 \tag{8.3}$$

Equation 8.3 is valid for kernels with size $k > 1$. When the kernel is 1×1, the number of parallel convolution windows coincides with the number of total convolution windows and, as a consequence, the number of iterations required to complete the whole convolution is equal to 1 as all the windows are non overlapping.
The advantage of using the presented *parallel non-overlapping data flow scheme* is the avoidance of data redundancy while preserving the parallel computation of convolution windows.

This data flow scheme can be directly and easily mapped on CLiMA. In fact, the idea is to store, in memory, the input feature map by assigning a pixel to each cell of the array. Weights are instead properly distributed to the cells and moved over the array in such a way that the convolution window shifting operation is reproduced and the parallel computation of non overlapping windows, as depicted in figure 8.13, is guaranteed. More details on weight distribution and how the computation of the convolution is managed in CLiMA will be given in subsection 8.3.3.

8.3.2 CLiM cell

The basic block of CLiMA is the CLiM cell. Its high level block diagram is represented in figure 8.14. The CLiM cell, as already explained in section 8.1,

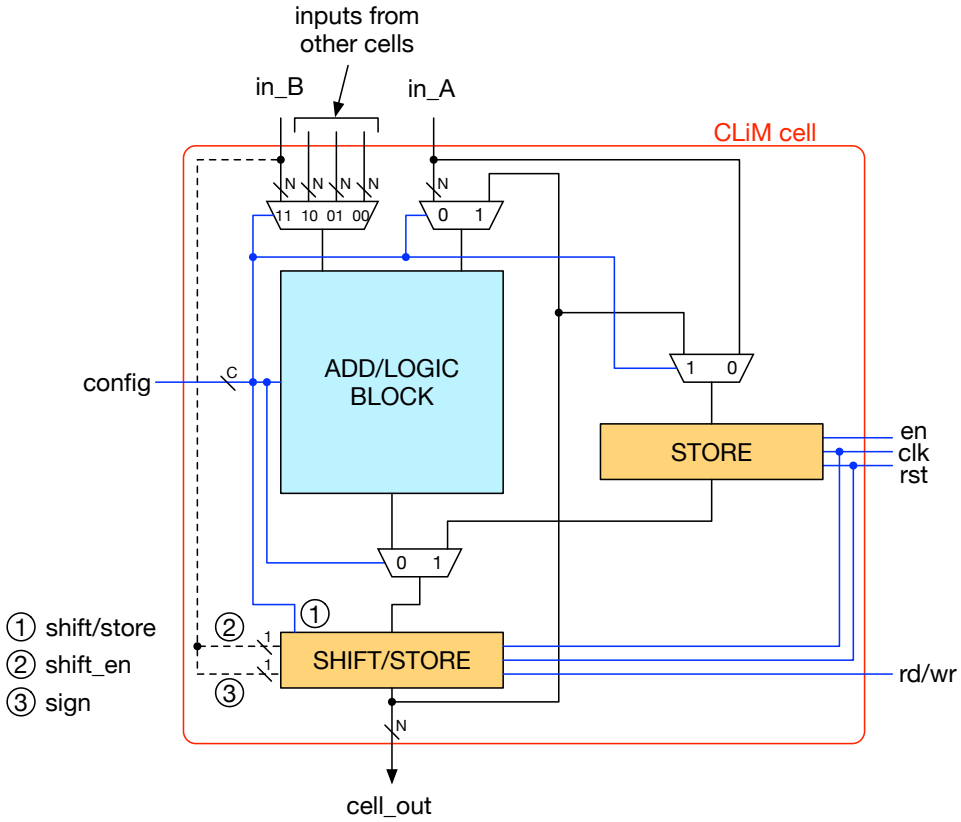

Figure 8.14: High-level block diagram of the N-bit CLiM cell.

integrates logic and storage. This cell has been designed in order to guarantee maximum *configurability* and *flexibility*. It consists of:

- an add/logic block that can be configured to perform arithmetic functions such as addition, subtraction and logic functions such as XOR, XNOR, AND, OR, NOT;
- a SHIFT/STORE block that can be used as a normal storage element or to execute shift operations on the data stored;
- a further STORE block that is used to hold partial or final results.

The content of the STORE block can be copied in the SHIFT/STORE block and viceversa. The CLiM cell in figure 8.14 is an N-bit cell made of N 1-bit cells that, properly connected together by exploiting inter-cell connections, create a complex

CLiM cell. The detailed internal structure of the CLiM cell is depicted in figure 8.15 (this is actually a simplified version of the real CLiM cell, its structure would be too confusing with all the details; here are reported the main ones). Row enable

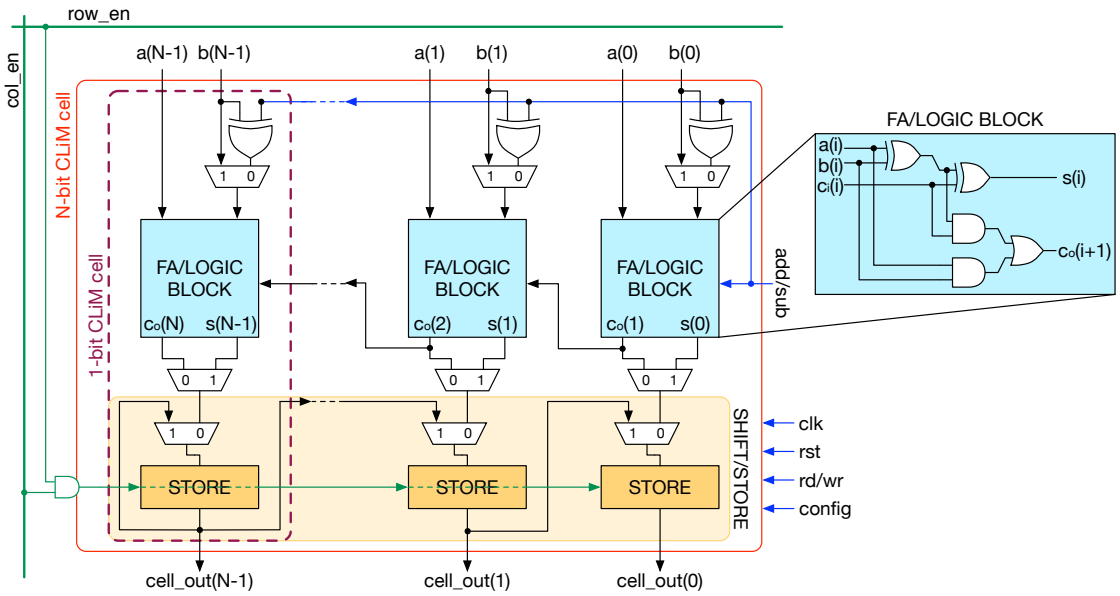

Figure 8.15: Detailed internal structure of the N-bit CLiM cell. By exploiting inter-cell connections, 1-bit cells are properly connected to create a complex CLiM cell.

and column enable lines are used to select cells in the CLiM array for read/write operations. Their role is equivalent to that of word-lines and bit-lines and, similarly, they are controlled by decoders (more details on decoders will be given in subsection 8.3.3). In figure 8.15, the leftmost 1-bit cell contains the MSB (most significant bit) while the rightmost cell contains the LSB (least significant bit). Each 1-bit CLiM cell is composed of two main elementary blocks: a 1-bit FA/logic block and a 1-bit store block. In addition to these components there is some other logic and the other store block for temporary/final results that, for sake of clarity, has not been drawn. The 1-bit FA/logic block is, basically, a simple full adder that can be also used to perform logic functions. The truth table of a full adder is reported in table 8.3. It can be observed that by fixing one or more full adder inputs to 0 or 1, the sum (S) and output carry (C_{out}) return different logic functions:

- when A = 0
  - $S = B \oplus C_{in}$ (XOR)
  - $C_{out} = B \cdot C_{in}$ (AND)
- when A = 1

A	B	C_{in}	S	C_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Table 8.3: Full adder truth table.

  - $S = \overline{B \oplus C_{in}}$ (XNOR)
  - $C_{out} = B + C_{in}$ (OR)
- when A = 0 and B = 1
  - $S = \overline{C_{in}}$ (NOT)
  - $C_{out} = C_{in}$ (indentity)
- when A = 1 and B = 0
  - $S = \overline{C_{in}}$ (NOT)
  - $C_{out} = C_{in}$ (indentity)

In order to support shift operations, store blocks are interconnected in a chain-like fashion with the output of a block being the input of the adjacent one. In order to support arithmetic right shifts[1], the output of the MSB CLiM cell is fed back to its input in order to replicate the sign bit. The structure shown in figure 8.15 supports only right shift operations. With few and simple modifications (not reported in figure for the sake of clarity) also left shifts can be handled inside the CLiM cell. Other logic elements inside the cell are:

- XOR gates used to calculate the 2's complement of input b when executing the operation $a - b$;
- different multiplexers used to select different signals, depending on the mode (storage, compute logic, compute arithmetic) in which the cell is being used.

[1]In computer arithmetic, a right or left shift is an operation that moves an N-bit operand by a given number of positions toward the right or left, respectively. When right shifting, if the shift is logical the leftmost positions are filled with zeroes, whereas if the shift is arithmetic the MSB, i.e. the sign bit, is replicated to fill the leftmost positions. When left shifting there is no difference in logic and arithmetic shift as in both cases the rightmost positions are filled with zeroes.

8.3.3 CLiM Array

The stucture of the CLiM array is shown in figure 8.7. What is missing in that scheme is a fundamental component of the architecture: the interconnections that allow data to be moved across cells for in-memory computation. In this specific case, interconnections have been designed specifically to support the convolution flow in the most efficient way possible. Inter-cells connections inside the CLiM array are depicted in figure 8.16. It can be seen that there are horizontal and vertical

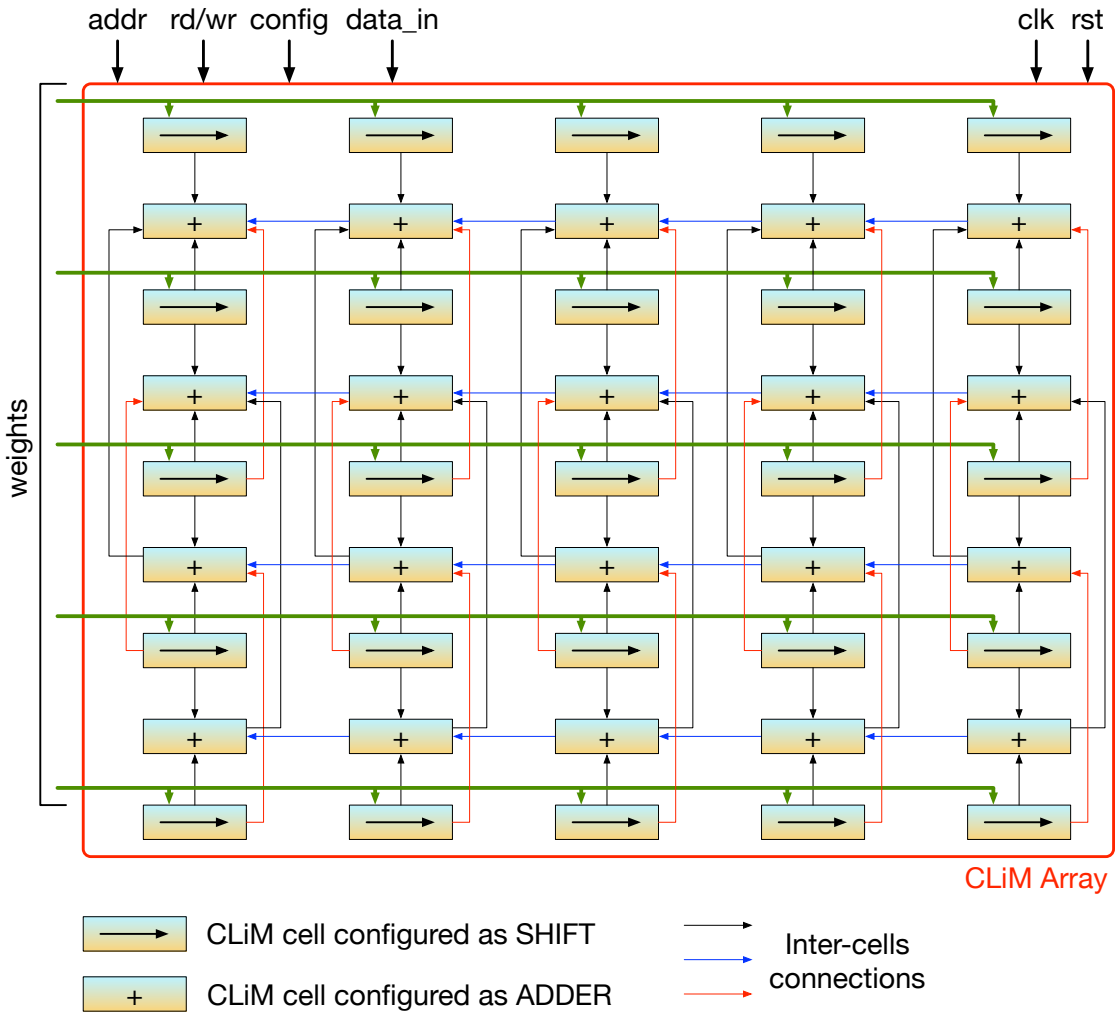

Figure 8.16: CLiM array structure. Inter-cells connections are shown: they are specifically designed to support the convolution flow. In addition, rows of the arrays are alternatively configured to be used as shift blocks or adders.

connections between neighboring and non CLiM cells. Moreover, the rows of the array are alternatively configured as shift blocks (even rows), which also receive weights from the external weight memory (refer to figure 8.7), or as adders (odd

rows). The idea is that even rows store input pixels and execute the shift operations, while odd rows execute the accumulations of the shifted pixels in order to produce the result of the convolution. Figure 8.17 shows how the computation of a 3×3 convolution window is managed inside the CLiM array. The very first step, which

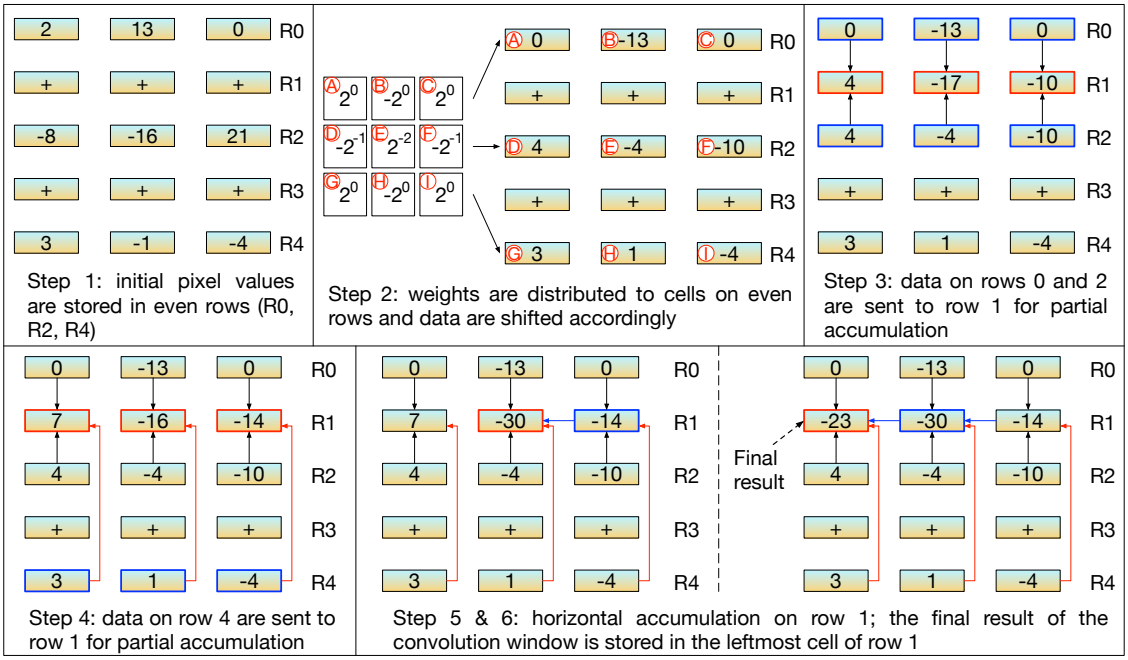

Figure 8.17: Convolution window computation inside the CLiM array.

can be considered a preliminary one, consists in loading input features into the even rows of the array. Then, weights are distributed across cells on even rows and data are shifted accordingly. Inside each cell the input feature is originally stored in the STORE block. At the beginning of the computation the data is copied from the STORE block into the STORE/SHIFT block where it will be shifted, while the original value of the pixel will be kept as is in the redundant STORE block. The reason of keeping the original value of the pixel in the STORE block is that each convolution layer in a CNN makes use of different weight kernels (high dimensional convolution), as explained in paragraph 1.3.2. This means that *more kernels share the same input feature map and CLiMA exploits this property in order to reuse data that are already inside the array.*

The example reported in figure 8.17 uses a 3×3 kernel. For example, the pixel whose correspondent weight is 2^0 is not shifted at all, whereas the pixel whose correspondent weight is -2^{-2} is be shifted by two positions to the right and, since the sign of the weight is negative, the sign of the pixel is also inverted. Once all data have been shifted, they are accumulated in the odd rows. In step 3, data on rows 0 (R0) and 2 (R2) are sent to row 1 where cells execute element-wise addition and update their content. In step 4, cells on row 4 sent their data to row 1 for

further accumulation. Finally, in steps 5 and 6 the partial results stored in row 1 are horizontally accumulated (from the right to the left) to obtain the final result of the convolution window. The final result is then copied in the STORE block of the cell.
The data flow explained since now makes use of the horizontal and vertical inter-cells connections to accumulate data and obtain the final result. However, when the size of the kernel varies (kernel dimensions vary across layers in a CNN) the number of cells involved in the computation of a convolution widow varies as well. The *interconnection fabric* designed and shown in figure 8.7 allows to *support any kernel size*, guaranteeing flexibility.

8.3.4 Control of CLiM Array

The control of the array is mainly managed by setting row/column decoders and masks properly. Row/column decoders are modified in order to enable more contiguous rows/columns at the same time. This is achieved by providing two addresses to the decoders: a starting address and an ending one, with the former being smaller than the latter. The decoders then activate all rows/columns included in the interval indicated by the starting and the ending address. However, in order to enable more complex patterns decoders are not enough. As shown in the example in figure 8.18, four convolution windows are executed in parallel in the CLiM array. In this case, the computation involves cells on rows and columns from 0 to 6. Hence, when setting row/column decoders the starting and ending addresses provided will be 0 and 6, respectively. This setting enables all rows and columns in the given interval of addresses. However, convolution windows are not adjacent and the third row and column should not be active. Row/column masks are used to mask the enabling signals coming from the decoders in order to disable the rows/columns inside an interval of addresses that are not involved in the computation.

8.3.5 Weights Dispatching

As shown in figure 8.7, a block called weight dispatcher is used to distribute weights over the array in order to reproduce the window shifting process typical of the convolution operation. Figure 8.19 describes how the weight dispatching works in CLiMA. Weights are distributed alternately among CLiM cells on each row. Cells 0 and 4 share the same weight as well as cells c1 and c5 and so on. As explained in subsection 8.3.1, a complete convolution operation is divided in different steps where non overlapping convolution windows are executed in parallel. Step after step, windows move over the input feature map (figure 8.13). Therefore, the goal is to reproduce the window shifting process by moving the convolution windows as showed in figure 8.19(A). In the CLiM array this movement is obtained by shuffling weights after each computation step (figure 8.19(B)). At time $t1$ 6 out of 7 cells

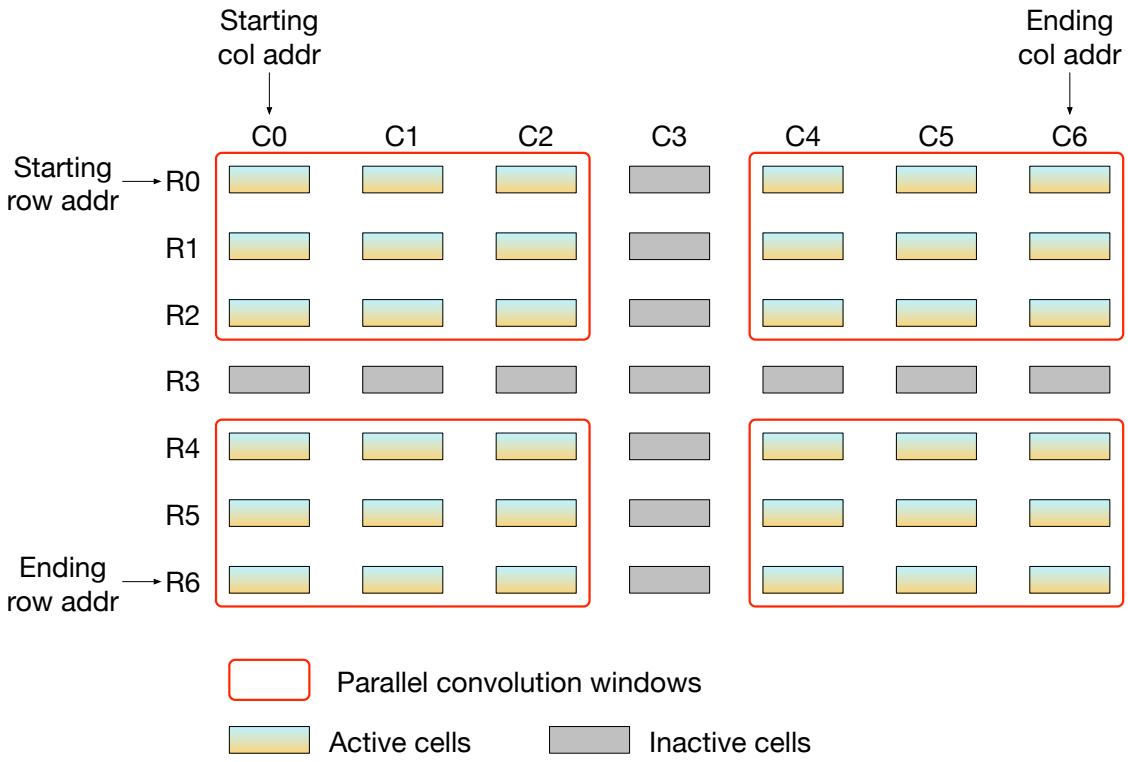

Figure 8.18: Example of four parallel convolution windows to be computed in-memory. The convolution windows might not be contiguous, hence, some rows/-columns between them are inactive. The combination of row/column decoders and masks is used to selectively enable/disable rows/columns.

in the row are active and weights are distributed in such a way that cells c0 and c4 receive weight w1, cells c1 and c5 receive weight w2 and so on. At the next computation step, time $t2$, weights are shuffled in order to move the convolution windows. At time $t3$ weights are shuffled again and convolution windows are shifted further. This dispatching mechanism is managed by the weight dispatcher block which, step after step, shuffles weights according to the window shifting pattern needed. Figure 8.19 depicts only an example of possible window shifting pattern. Since the dispatching mechanism is flexible, different patterns (when varying the stride, as instance) can be mapped. Moreover, the dispatching mechanism has been optimized to manage 3×3 filters since this is the most widely used kernel size. Nonetheless, smaller or larger kernels can be easily managed as well. In fact, another common kernel size is 1×1 and, in this case, the same kernel would be distributed to all rows with no need to shuffle weights as all convolution windows can be computed in parallel at the same time (no overlapping regions when the kernel is 1×1). Other common sizes are, for example, 5×5 or 7×7 filters (even though filters larger than 3×3 are usually used very few times and only in the first

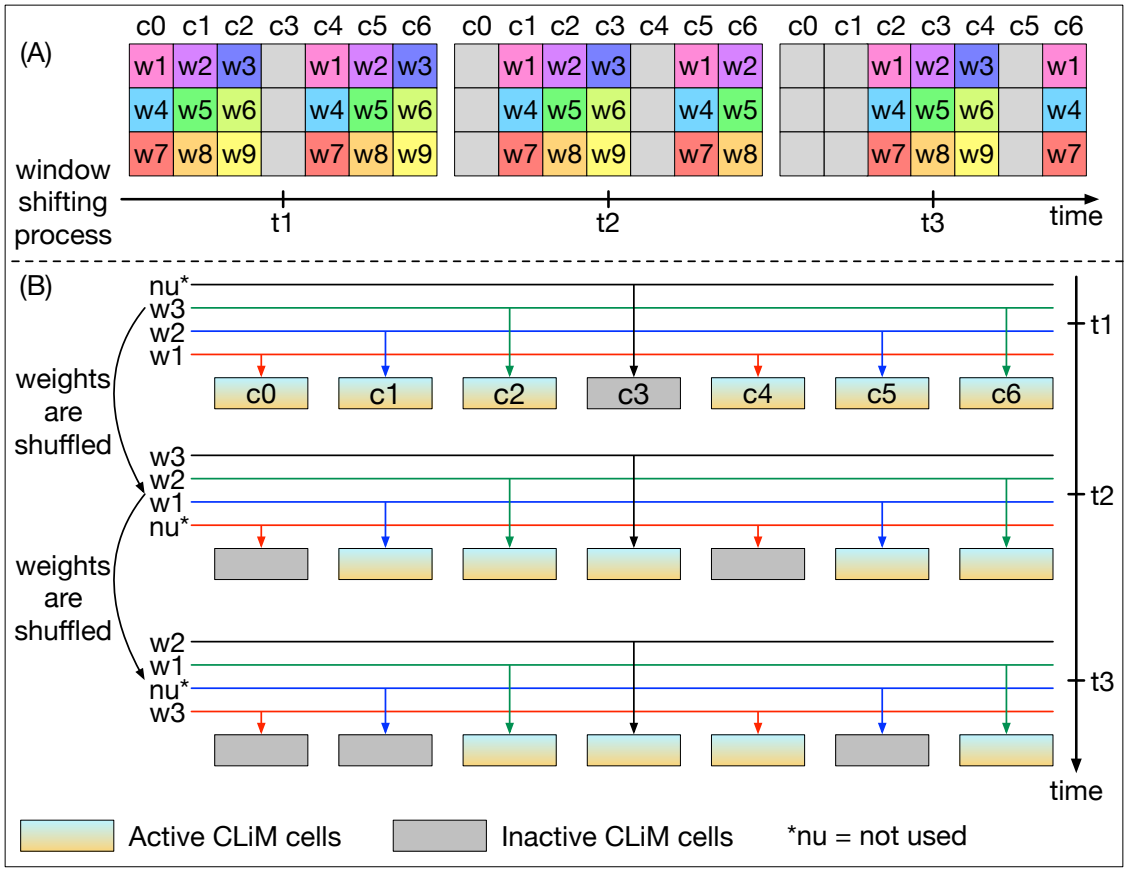

Figure 8.19: Weight dispatching mechanism in CLiMA. (A) Window shifting process. (B) The weight shuffling process managed by the weight dispatcher block allows to reproduce inside the CLiM array the window shifting mechanism used in the convolution.

layer of the network). In this case the weight distribution would be more complex as the convolution window would be processed in more than one step.

8.3.6 Data Reuse Possibilities in CLiMA

One of the main advantages of using a LiM architecture such as CLiMA for convolution processing is the possibility to reuse data computed inside the array for further processing without the need to move it outside, as it is done in architectures that do not exploit the LiM principle. When targeting CNNs, different data reuse possibilities can be delineated in CLiMA, as also depicted in figure 8.20. Let suppose to map each channel of the input feature map on a different CLiM array (figure 8.20(A)). As explained in paragraph 1.3.2, the convolution operation is high-dimensional and it involves multiple filters that share the same input feature map

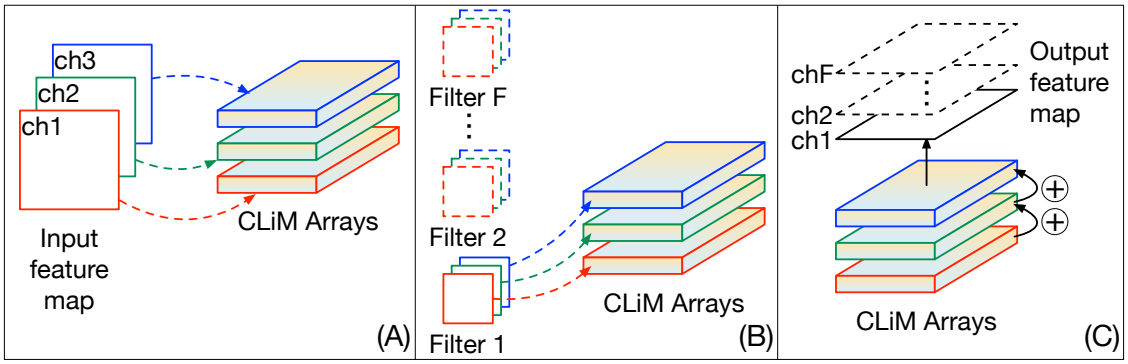

Figure 8.20: Data reuse possibilities. (A) Each channel of the input feature map can be mapped on a CLiM array. (B) The input feature map stored in the CLiM arrays is reused by different kernels in order to compute high-dimensional convolution. (C) After convolution, each CLiM array stores partial output results that are accumulated across CLiM arrays to obtain the final output feature map.

to produce an output feature map with as many channels as the number of filters. This means that the input feature map stored in CLiMA can be reused by different filters (figure 8.20(B)). Each channel of the filter is assigned to the correspondent channel of the input feature map and in-memory convolution is performed. Each CLiM array will now store the partial results of the convolution. These partial results can be accumulated across the arrays (figure 8.20(C)) in order to produce the final output feature map.
So, the type of data reuse in CLiMA are:

- *filter reuse*: filters are reused across the input feature map to perform convolution based on the sliding window process;
- *input feature map reuse*: input feature maps are reused by several filters to perform high-dimensional convolution;
- *partial results reuse*: partial results obtained by in-memory convolution are reused, inside CLiMA, for further processing to obtain the final output feature map.

The filter and feature map reuse properties are typical of the CNN data flow, as explained in section 1.5, and CLiMA intrinsically exploits them in order to reduce data movement and memory accesses. In addition, since the convolution operation is completely performed in CLiMA, partial results are already stored in memory, hence, there is no need to move them and they can be directly reused for further processing to obtain the final result.
On the contrary, what is usually done in non-LiM architectures is:

1. the input feature map and the first filter are *read from the memory*;

2. convolution is performed inside an execution unit and partial results are *written back in memory*;
3. once the convolution is completed all partial results are *read from the memory*;
4. accumulation of partial results is performed and final results are *written into the memory.*
5. back to step 1 until all filters have been used.

It is clear that, thanks to the *intrinsic data reuse possibilities* of CLiMA, the advantage of using such architecture to perform convolution is huge in terms of *reducing memory accesses.* As it will be shown in the next subsection, this is not the only advantage as the *intrinsic massively parallelism* offered by CLiMA allows to *speed up the convolution considerably.*

8.3.7 Results and Comparison

The architecture of CLiMA was conceived to be technology independent, therefore, it is not bounded to any specific technology because the main aim is to validated that the CLiMA computational model is effective with respect to a conventional one[2]. CLiMA was modeled by using a *fully parametric VHDL code* and validated through extensive simulations and by comparing the results obtained from the VHDL description with an analogous model developed in MATLAB.
In order to validate the effectiveness of the CLiMA, the architecture has been compared to the Deep Learning Processor presented in chapter 3.
For this purpose, an *analytic computational model* of CLiMA was defined. The parameters taken into account by this model are:

- the characteristics of the convolution layer, i.e. input feature map dimensions (W_{in}, H_{in}), kernel dimensions (k), stride (S) and output feature map dimensions (W_{out}, H_{out});
- number of non-overlapping convolution widows that can be executed in parallel (according to the parallel non-overlapping data flow scheme described in subsection 8.3.1) in each layer;
- cycles needed to execute the convolution of a single window.

[2]In this contest, a *conventional architecture* or *conventional computational model* is referred to a system where the processing unit and the memory are physically separated and data are moved from the memory into the processing core for computation and results are then written back into the memory.

In order to compute the number of non-overlapping convolution windows that can be executed in parallel, the characteristics of the convolution layer must be taken into account. Starting from the output feature map size, the total number of convolution windows needed to complete the convolution of a layer is equal to the size of the output feature map that can be calculated according to the following equation:

$$W_{out} = H_{out} = \frac{W_{in} - k}{S} + 1 \tag{8.4}$$

Input and output feature maps are always square, therefore they have the same width ($W_{in/out}$) and height ($H_{in/out}$). Since, a single convolution window produces an output pixel (figure 1.13), the total number of convolution windows (CW_{tot}) is exactly equal to the total number of output pixels:

$$CW_{tot} = W_{out} \cdot H_{out} = W_{out}^2 = H_{out}^2 \tag{8.5}$$

The number of non-overlapping convolution windows on a layer can be computed as:

$$CW_{non-ov} = \left(\frac{W_{in}}{k + (S-1)} \right)^2 \tag{8.6}$$

In order to compute a complete convolution by using the parallel non-overlapping data flow scheme, more computation rounds are needed as shown in figure 8.8. The number of rounds is simply given by the upper bound of the ratio between the total number of convolution windows (equation 8.5) and the number of non-overlapping ones (equation 8.6):

$$C_{rounds} = \left\lceil \frac{CW_{tot}}{CW_{non-ov}} \right\rceil \tag{8.7}$$

Now, in order to compute the number of cycles required to execute the complete convolution on a layer, the single convolution window must be first taken into account. The number of cycles required to execute a single convolution window depends on the size of the window, hence, on the size of the kernel. Considering how the computation of a $k \times k$ convolution window is managed inside the CLiM array (as shown in figure 8.17), the number of cycles for a single convolution window (CW_{cycles}) are given by:

$$CW_{cycles} = 8 + 1 + \left(\frac{k-1}{2} \right) + (k-1) \tag{8.8}$$

The terms that constitute equation 8.8 are:

- number of cycles needed to perform shifts: it is a constant value because weights are on 8 bits hence, in the worst case, 8 shifts are required. In CLiMA data is shifted of 1 bit per cycle.

- number of cycles needed to perform accumulations:
  - 1 cycle to accumulate data pairs as shown in step 3 of figure 8.17; in this case the number of cycles in independent from the size of the kernel because these accumulations can always be done in parallel;
  - $(k-1)/2$ cycles to perform accumulations of non-adjacent data as shown in step 4 of figure 8.17; this figure shows only the case of a 3×3 kernel, however, when the size of the kernel increases the number of cycles required to perform non-adjacent accumulations increases because the convolution window is larger and there are more non-adjacent data to accumulate;
  - $k-1$ cycles to perform final horizontal accumulations as shown in steps 5 and 6 of figure 8.17; as well as for the previous term, also in this case the number of cycles grows as the size of the kernel, hence the convolution window, increases because there are more horizontal accumulations to perform.

The total number of cycles (C_{cycles}) needed to execute a complete convolution on a layer is given by:

$$C_{cycles} = CW_{cycles} \cdot C_{rounds} \tag{8.9}$$

In a single convolution round CW_{non-ov} windows are executed in parallel. Knowing that the number of cycles taken to execute them is CW_{cycles}, the total number of cycles to complete the convolution of a layer is expressed by equation 8.9.
Two CNNs were used to extract results and perform comparisons with the Deep Learning Accelerator: AlexNet [69] and ResNet-18 [48]. For what concerns the Deep Learning accelerator, the assumption is that each convolution window is assigned to a processing element for parallel execution. The number of cycles to execute a convolution window depends on the kernel size, hence $k \times k$ because the DL Accelerator has a throughput of 1 MAC operation per cycle (refer to the throughput model in subsection 4.4.2). Regarding CLiMA, the assumption is that a certain number of non-overlapping windows are executed in parallel (according to the proposed parallel non-overlapping data flow scheme). Four different scenarios were considered; what varies between one scenario and the other is the parallelism level, i.e. the total number of parallel processing elements in the DL processor and the total number of parallel non-overlapping windows in CLiMA. Figures 8.21 show the number of cycles required by CLiMA to perform the convolution in AlexNet. In both graphs, on the x axis are reported the five convolution layers of AlexNet while on the y axis the number of cycles are shown. The four different lines represent the different scenarios considered. The blue line is the worst scenario, with only 10 parallel non-overlapping convolution windows, whereas the red line represent the best scenario with a window parallelism of 60. The graph in figure 8.21(B) is a magnification of the left graph, showing more in detail the number of cycles for the

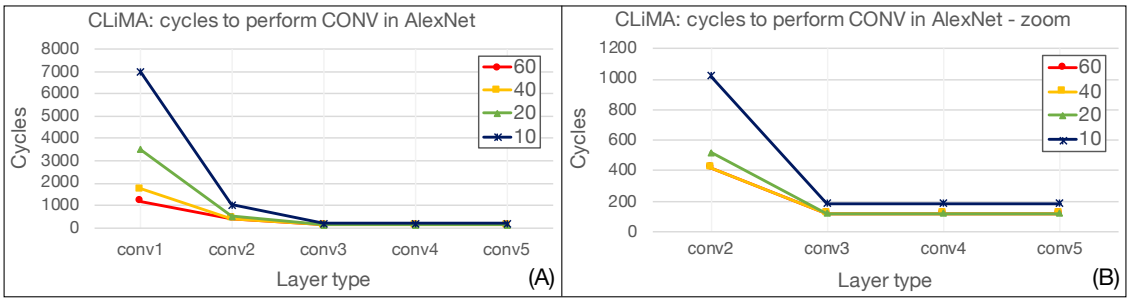

Figure 8.21: (A) Cycles to execute convolution in CLiMA considering AlexNet as workload and four different parallelism scenarios. (B) Detail of the last four layers of AlexNet.

last four convolution layers of AlexNet. As expected, by increasing the parallelism (and the dimension of the CLiM array, as a consequence) the number of cycles to complete the convolution reduces. This reduction is drastic in the first layer of the network while it is less evident in the last three layers. This behavior depends on the characteristics of the layers. In particular, the first and second layers have a large size (W_{in}, H_{in}), which means that the number of parallel non-overlapping windows is high as well. Going deeper in the network the size of the layers decreases as well as the number of non-overlapping windows until a lower limit is reached where having more parallelism does not influence the number of cycles.
The behavior is similar for the DL Accelerator, as highlighted in the graphs in figure 8.22. The four different lines refer, as before, to the four different parallel

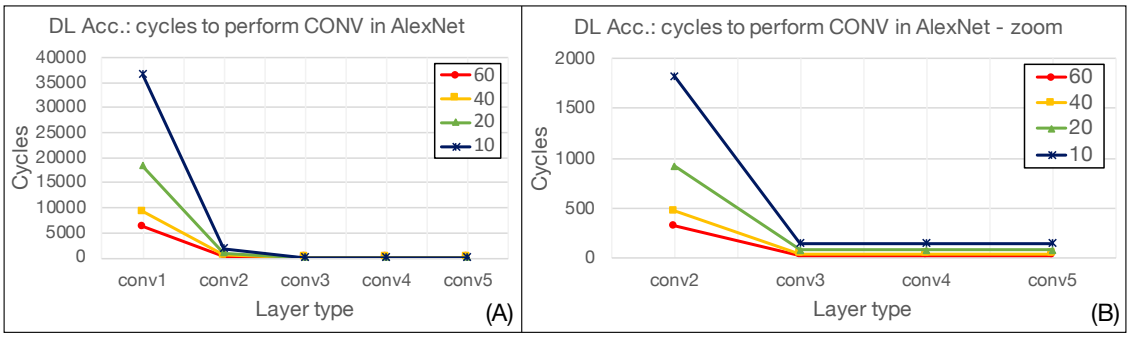

Figure 8.22: (A) Cycles to execute convolution in the DL Accelerator considering AlexNet as workload and four different parallelism scenarios. (B) Detail of the last four layers of AlexNet.

scenarios with the blue line representing the worst scenario (only 10 parallel PEs) and the red line representing the best one (60 parallel PEs). As for CLiMA, also in this case when going deeper in the CNN, the size of the layers decreases as well as the number of convolution windows. For this reason, as clearly highlighted in

the graph in figure 8.22(B), increasing the parallelism does not have a big impact in the total number of cycles taken to complete the convolution.
In order to compare CLiMA and the DL Accelerator the average number of cycles required to complete the convolution in the four different scenarios have been calculated. The results are shown in the graph in figure 8.23. The term conventional refers to the DL Accelerator. The x axis reports the parallelism level

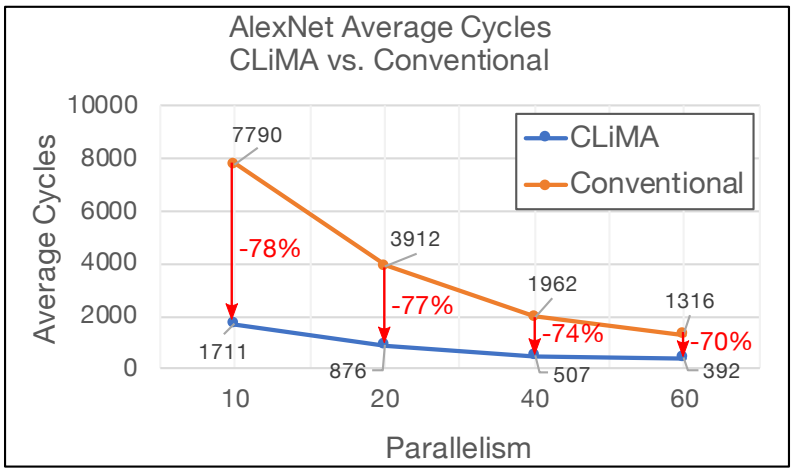

Figure 8.23: AlexNet: number of average cycles to perform convolution in different parallelism scenarios. CLiMA and the DL Accelerator (referred to as conventional) are compared.

(non-overlapping convolution windows for CLiMA, parallel PEs for the DL Accelerator). The y axis reports the average cycles. It can be clearly seen that in all the scenarios CLiMA outperforms the DL Accelerator. The percentage reduction in terms of average of cycles that CLiMA provides is quite significant. Indeed, it is equal to 70% when considering a parallelism of 60 and steadily increasing while the parallelism decreases, with a peak reduction of 78% when considering the smallest parallelism scenario. The cycles reduction is more evident when the parallelism is smaller, which further proves the effectiveness of the CLiMA computational flow.
Similar trends can be observed when taking as workload another CNN, ResNet-18. Figure 8.24 reports the results of performing convolution in CLiMA (graphs in figures 8.24(A) and (B)) and in the DL Accelerator (graphs in figures 8.24(C) and (D)). As for AlexNet, when going deeper in the network, layers tend to shrink and the number of convolution windows reduces, that is why increasing the level of parallelism in both architectures does not have any significant advantages in terms of performance. The average cycles for all layers in ResNet-18 has been computed as well and the results for CLiMA and the DL Accelerator are depicted in figure 8.25. Also in this case CLiMA outperforms the DL Accelerator. In fact, it can be seen that CLiMA provides a percentage reduction in terms of cycles, when compared to the DL Accelerator, that ranges from 45% when considering the maximum

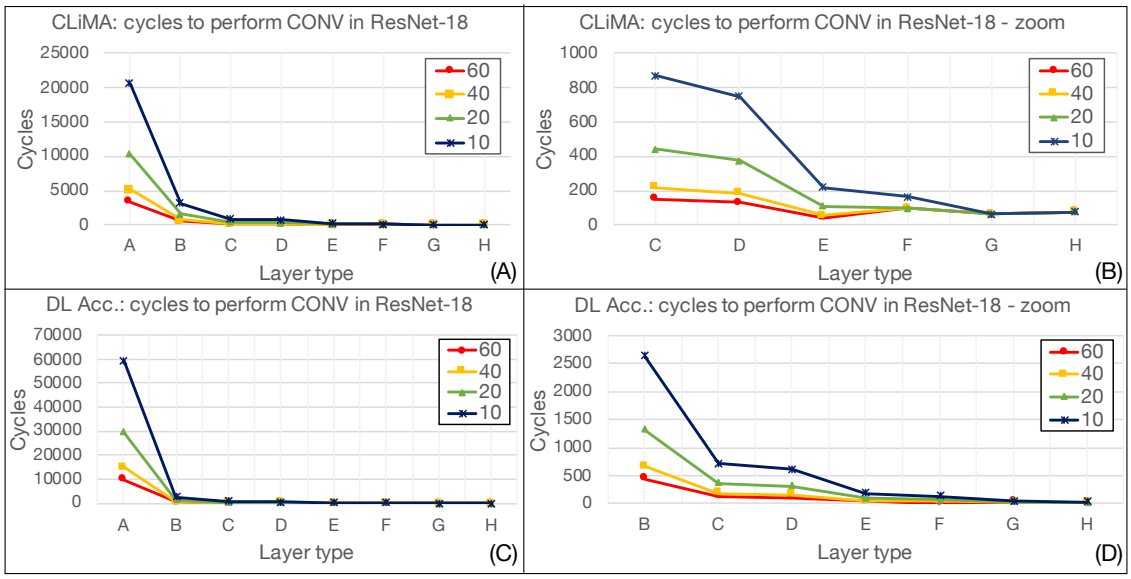

Figure 8.24: (A) Cycles to execute convolution in CLiMA considering ResNet-18 as workload and four different parallelism scenarios. (B) Detail of the last layers of ResNet-18. (C) Cycles to execute convolution in the DL Accelerator considering ResNet-18 as workload and four different parallelism scenarios. (D) Detail of the last layers of ResNet-18.

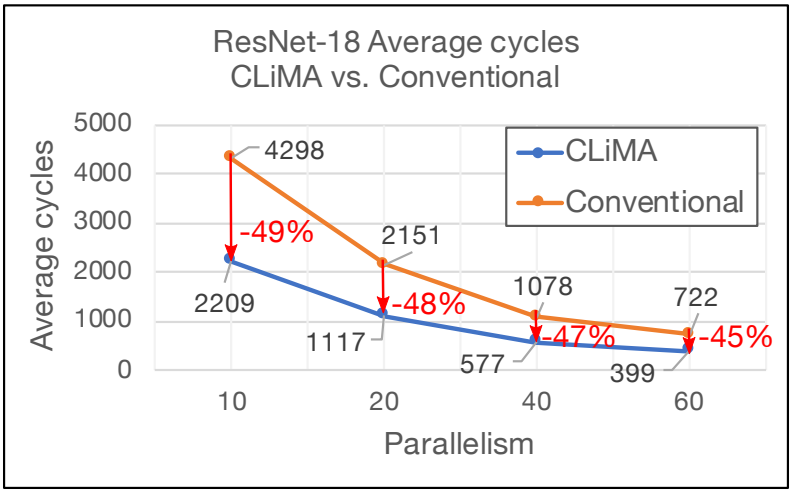

Figure 8.25: ResNet-18: number of average cycles to perform convolution in different parallelism scenarios. CLiMA and the DL Accelerator (referred to as conventional) are compared.

parallelism, to 49% when considering the minimum one. In this case the percentage reduction is smaller than in the AlexNet case. This depends on the characteristics of the network (i.e. size of the layers and of the kernels, stride) and, in particular,

on how many non-overlapping convolution windows each layer has. The more they are the more the reduction of cycles provided by CLiMA.
Even if described to be technology independent, the CLiM array has been synthesized on a 28nm FDSOI technology (the same used to synthesize the DL Accelerator) in order to have an estimation of the clock frequency at which the array can work. The size of the array synthesized is such that 10 non-overlapping windows can be executed in parallel. The same case has been considered for the DL Accelerator, with 10 PEs working in parallel. In both cases the working frequency is close to 1.8 GHz. This data, together with the average cycles calculated before in case of a parallelism equal to 10 are used to estimate the execution time taken by CLiMA and the DL Accelerator to perform the convolution. The results are reported in table 8.4. As expected, at the same working frequency the average execution time

CNN type	**Architecture**	**Average cycles**	$\mathbf{T_{exec}}$ **[µs]**
AlexNet	CLiMA	1711	0.95
	DL Acc.	7790	43.2
ResNet-18	CLiMA	2209	1.2
	DL Acc.	42939	24

Table 8.4: Performance estimation of CLiMA vs. the DL Accelerator for AlexNet and ResNet-18. The parallelism is fixed to 10 parallel convolution windows for CLiMA and PEs for the DL Accelerator. The target clock frequency is 1.8 GHz for both architectures.

in CLiMA is lower than in the DL Accelerator.
The main point of the comparison is to demonstrate that exploiting the Logic-in-Memory paradigm it is possible to reduce the number of data exchanges between the processing unit and the memory. In case of CLiMA, the processing unit is represented by the CLiM array while the external memory is represented by the weights memory. This memory is accessed only once to read the weights which are then reused to execute all the convolution windows on the whole input feature map. This means that only $k \times k$ reading operations from the weights memory are required. The input features are already inside the CLiM array. The same is true for the output results which are computed inside the array. Therefore, no write operations to an external memory are required.
For what concerns the DL Accelerator, the external memory is represented by the input and output buffers in the PE. In this case both input features and weights are read form the input buffer and moved to the execution unit. The output results for each convolution window are then written in the output buffer. This means that for each convolution window the number of memory accesses required is:

- $(k \times k) + (k \times k)$ to read weights and correspondent input features from the

input buffer;

- $W_{out} \cdot H_{out}$ to write the output results in the output buffer.

It is important to underline that in this evaluation it is not being considered the presence of a main memory (external to CLiMA or the DL Accelerator) form which weights and input features are loaded and written into the input buffer of each PE inside the DL Accelerator or into the CLiM array. These operations are always required and cannot be avoided even when considering a LiM-based architecture (because it needs to loaded with data somehow). Moreover, CLiMA and the DL Accelerator are very different from each other under many aspects, hence, comparing them is not easy and the main risk is to make a non fair comparison. In order to avoid this, the evaluation is carried out when considering the basic case of the convolution computation, that is a single kernel shifted on a single input feature map. Results for AlexNet and ResNet-18 are reported in figures 8.26 and 8.27. In both cases, thanks to the data reuse possibilities that CLiMA offers and

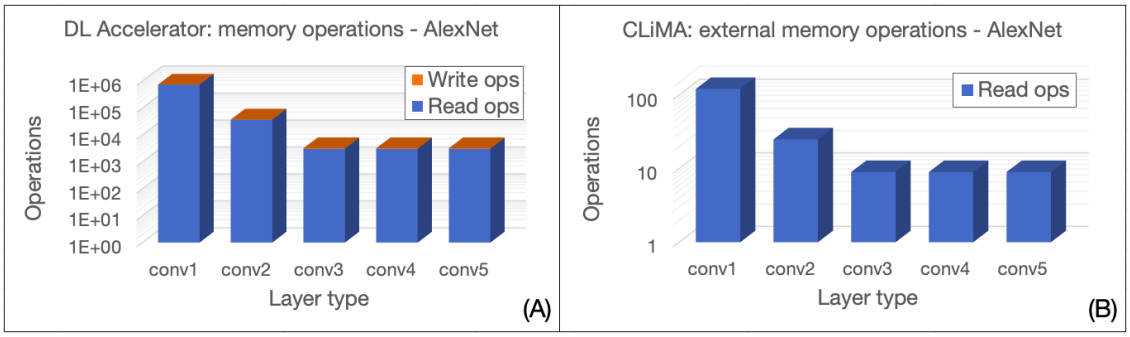

Figure 8.26: AlexNet: (A) DL Accelerator memory accesses (read operations to retrieve weights and input features and write operations to store final results). (B) CLiMA: external memory accesses. Only read operations are performed to retrieve weights from the weight memory.

thanks to the LiM computing paradigm, memory accesses are drastically reduced with respect to a conventional computing paradigm such as the one used in the DL Accelerator.

8.4 CLiMA: Strong Points and Issues

In this section, a recapitulation of CLiMA is given in the perspective of highlighting strong points and issues. The main strong points of the proposed architecture can be identified in the following.

- In-memory computation: being a Logic-in-Memory architecture, data processing in CLiMA is done directly inside the memory without the need to

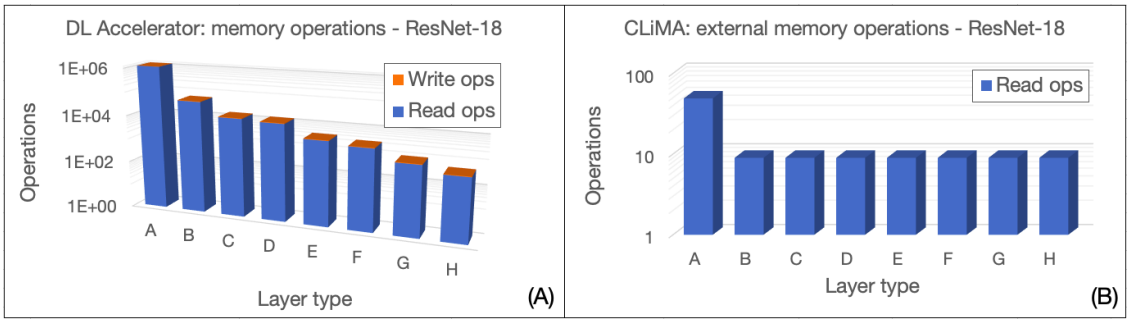

Figure 8.27: ResNet-18: (A) DL Accelerator memory accesses (read operations to retrieve weights and input features and write operations to store final results). (B) CLiMA: external memory accesses. Only read operations are performed to retrieve weights from the weight memory.

move data. The reuse of data already present in memory is maximized when further processing is needed.

- Configurability: the CLiM cell, which is the basic LiM element of CLiMA, can be configured to perform logical and arithmetic in-memory operations by reusing the same simple logic.
- Flexibility: inter-cells connections can be exploited to build more complex in-memory functions, making CLiMA a flexible architecture that can be used for different applications.
- Parallelism: CLiMA is intrinsically parallel as its main core is the array, a collections of cells that can work together and in parallel. This feature is exploited to accelerate data-intensive applications.

For what concerns issues, two main ones can be identified. Firstly, the architecture proposed in section 8.3, although configurable and flexible in terms of operations that can be executed in memory, it is not in terms of inter-cells connections, hence, possible computation patterns. In fact, the interconnection fabric presented in subsection 8.3.3 has been specifically designed to maximize the CNN data flow. As it will also be clear from chapter 9 (exploration of other CLiMA approaches targeting other applications), inter-cells connections vary quite a lot depending on the target application since the data flow changes as well. This means that, some algorithms may require certain inter-cells connections, to be executed in an efficient way, that other algorithms may not need. The non-generality of inter-cells connections limits the flexibility of CLiMA. The second issue is related to the control of CLiMA, in particular, to the management of operations and data flow inside the CLiM array. As depicted in figure 8.17, the convolution window computation consists of different steps in which different cells interact, exchange data and perform computation.

Moreover, the active cells involved in the computation vary step after step (figures 8.18 and 8.19). The management of the computation flow inside the array is not at all trivial especially because the data exchange between cells must be carefully coordinated in order to avoid that cells send or receive wrong data.

8.5 Beyond CMOS: CLiMA for pNML

As explained in chapter 6, beyond-CMOS technologies are being studied by the research community as possible substitutes of CMOS [81]. Among these, Nano Magnetic Logic (NML) is one of the most promising as it combines non volatility with computing capabilities, 3D integrability and low power consumption. There exists different implementations of NML technology and one of the most interesting is perpendicular NML (pNML) [10]. In this section it is presented the design of the CLiM cell (different versions) and of the CLiM array, fully based on pNML technology.

8.5.1 pNML: perpendicular Nano Magnetic Logic

The basic element of Nano Magnetic Logic is the nanomagnet, a multi-layer Co/Pt stack small enough (in the order of tens of nanometer) to guarantee the presence of a single magnetic domain. Nanomagnets are bi-stable elements: in fact, thanks to magnetization anisotropy, only two stable magnetization states are allowed. In pNML, nanomagnets are characterized by perpendicular (with respect to the magnet plane) magnetization anisotropy (reason why the technology is called *perpendicular* NML) and the direction of the magnetization, as shown in figure 8.28(A), is used to encode logic values '0' and '1'. Differently from CMOS, in pNML technology the information is driven by magneto-static field-coupling interaction among the nanomagnets. Logic gates and interconnections can be easily built by linking nanomagnets in a proper way. The field-coupling interaction among pNML cells enables the transfer of magnetic charge, hence, the signal transmission. A specific direction of the signal propagation is obtained by modifying the local magnetic properties of nanomagnets (figure 8.28(B)) [18]. In particular, by exploiting Focused Ion Beam (FIB) irradiation, a region of the nanomagnet is made more sensitive to magnetic field variations. This induced high sensitive region is called Artificial Nucleation Center (ANC). The ANC can be considered as the input of the magnetic device; in other words, it is the point where a domain wall starts nucleating and, eventually, it propagates through the nanomagnet reversing its initial magnetization direction (figure 8.28(C)). A recent research development [67] modifies the way in which ANCs are created: instead of using FIB irradiation, ANCs can be created by changing the shape and thickness of one side of the nanomagnet (figure 8.28(E)).

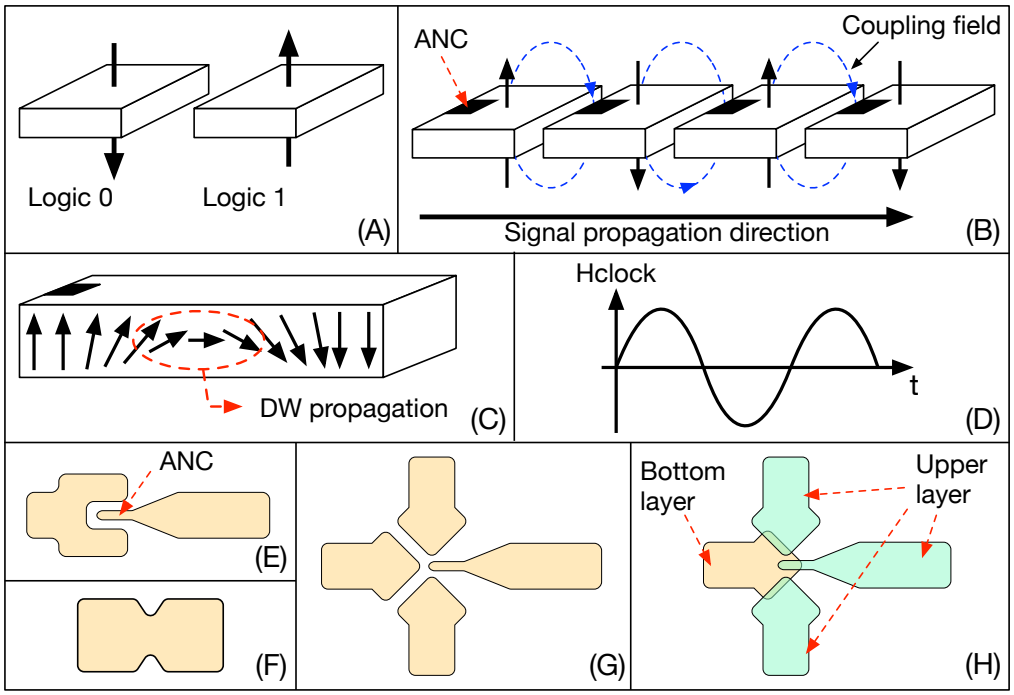

Figure 8.28: pNML technology basics. (A) Single-domain nanomagnets are bistable devices. The magnetization direction is used to encode the binary values 0 and 1. (B) The Artificial Nucleation Center (ANC) produced by FIB irradiation guarantees that the signal propagation direction is unidirectional. (C) The ANC is the point where a domain wall is nucleated and eventually propagated inside the nanomagnet, causing the switching of the magnetization direction. (D) Global out-of-plane magnetic field used as clocking mechanism in pNML circuits. (E) Inverter. (F) Minority voter. (G) 3D minority voter.

The propagation of information in a pNML circuit can be achieved by applying a global out-of-plane magnetic field (figure 8.28(D)) [11] that has the same function of the clock signal in a CMOS circuit. This external magnetic field is sinusoidal and its intensity is such that it can force the switching of the magnetization direction of the nanomagnets. So, the combined action of the external magnetic field and ANCs allows a correct transmission of the information in a defined direction. As shown in figure 8.28(B), the coupling field (dashed blue line) of a magnet has an effect on the ANC of its neighboring nanomagnet and influences the magnetization switching. In pNML circuits nanomagnets tend to arrange themselves in an anti-parallel (figure 8.28(B)) or parallel way, depending on their relative position.
Figure 8.29 shows how the clocking mechanism work in pNML circuits and how information is propagated through a pNML wire (i.e. a chain of nanomagnets). The pNML wire has one input magnet (IN) whose magnetization direction is fixed at logic 1. At time $t = 0$ there is no external magnetic field applied. At time

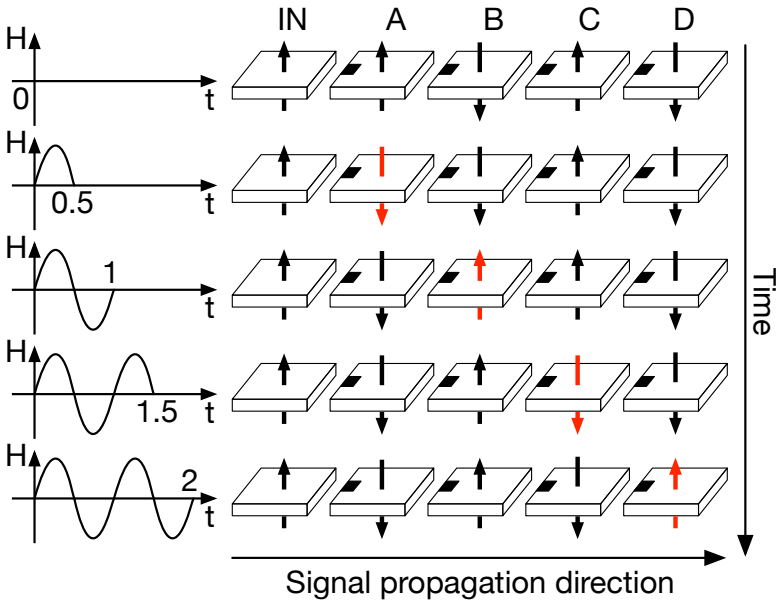

Figure 8.29: Clocking mechanism in pNML technology.

$t = 0.5$ a positive field is applied. Since it has the same magnetization direction of the input nanomagnet, the two contributions superpose and cause the switching of magnet A, in anti-parallel direction. In the next semi-period ($t = 1$) the external field and magnet A have the same direction and magnet B switches accordingly. In this way a logic 1 is propagated through the chain.
The basic elements to build pNML logic circuits are the inverter (figure 8.28(E)), the notch (figure 8.28(F)) and the minority voter (figure 8.28(G), 3D version in figure 8.28(H)), a function whose output is equal to the minority of the inputs. A *notch* is a shape deformation that acts as a barrier, blocking the signal propagation until it is depinned by means of a short in-plane magnetic field pulse [41]. Notches can be used to create magnetic memory elements [90][36].
One of the main advantages of pNML technology is that it allows to create 3D structures [9][33][35][34] with nanomagnets distributed on different planes. This fundamental characteristic, together with the non-volatility of the NML technology, allows to integrate memory and logic in the same device, perfectly *embodying the concept of Logic-in-Memory*. Previous works on the exploration of 3D NML structures were presented in [24][89][38][36][99]. However, none of these works propose a complete Logic-in-Memory structure, which is instead the aim of what will be presented in the following.

8.5.2 MagCAD: from layout to VHDL

The designs that will be shown in the next subsections were created using MagCAD [88], an enhanced graphic editor software for emerging technologies fully developed at the VLSI laboratory (research group of the Department of Electronics and Telecommunications of Politecnico di Torino).MagCAD can be used to *design and validate* custom NML circuits.
The user can create its own design by means of a simple and intuitive GUI (Graphical User Interface) that provides a layout workspace and building blocks (i.e. magnet, notch, inverter and others) that can be dragged and dropped inside it. Building blocks can be connected and combined together to form more complex structures. Moreover, MagCAD supports both *planar* and *3D designs*, giving the user the possibility to add multiple layers and place magnets on them.
In addition, MagCAD supports *hierarchical design*; in fact, the layout can be exported and used as a component in other designs to build complex architectures. Another key feature of MagCAD is that it supports the *generation of the VHDL description* of a circuit starting from its layout. The generated VHDL files can be directly used, together with a testbench, to simulate the behavior of the entity using a standard HDL simulator. When exported, MagCAD generates also a log file that contains useful information such as the number of VHDL items created or an estimation of the area occupied by the NML design.
The VHDL description generated by MagCAD makes use of a *compact VHDL model* [115] developed for pNML devices, based on physical models and technological parameters extracted from experimental results. As of now, different micro-magnetic simulators (e.g., mumax^3, OOMMF, NMAG) can be used to perform low-level simulations of magnetic devices and structures but, one of their main limitations is the size of the structure that can be modeled and simulated. In fact, low-level micromagnetic simulations are extremely time and resource consuming, hence, simulating large and complex structures is not feasible as it would require too much time and resources. On the contrary, MagCAD offers the possibility to design, simulate and validate even very complex structures.

8.5.3 pNML CLiM Cell and Array

Two different pNML-based versions of the CLiM cell were designed. Both of them are multi-layer structures.
The first pNML CLiM cell, shown in figure 8.30, is the most complex one and it is the exact pNML-equivalent of the cell presented in subsection 8.3.2. The design is three-dimensional with magnets placed on four different layers. The main building blocks of the cell are the computational element (figure 8.31(A)) and the memory cell (figure 8.31(B)). The two building blocks are implemented on three layers. The computational element (figure 8.31(A)) is a 1-bit configurable full adder that can

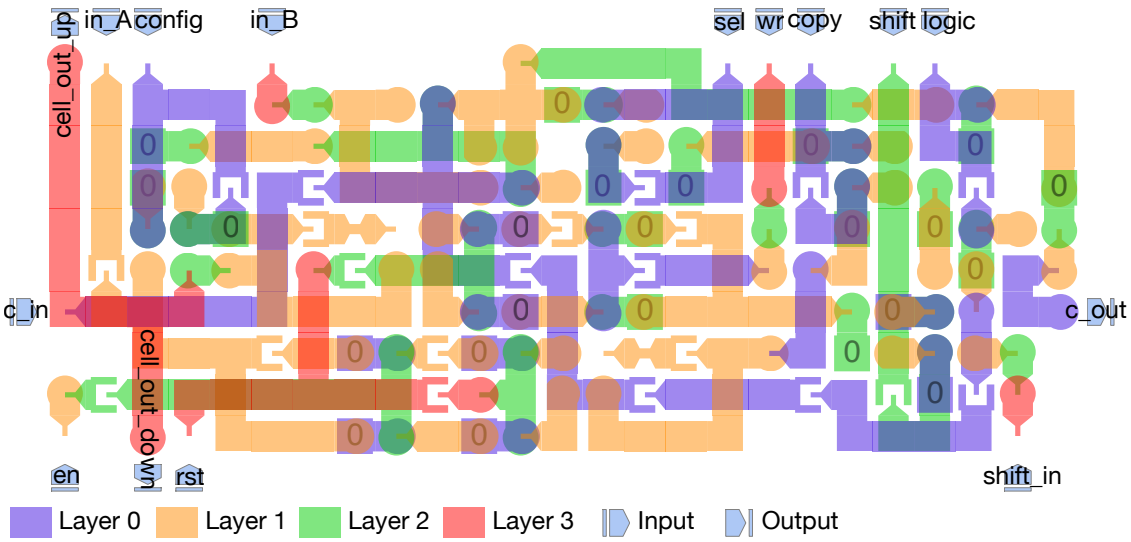

Figure 8.30: pNML CLiM cell: complex version.

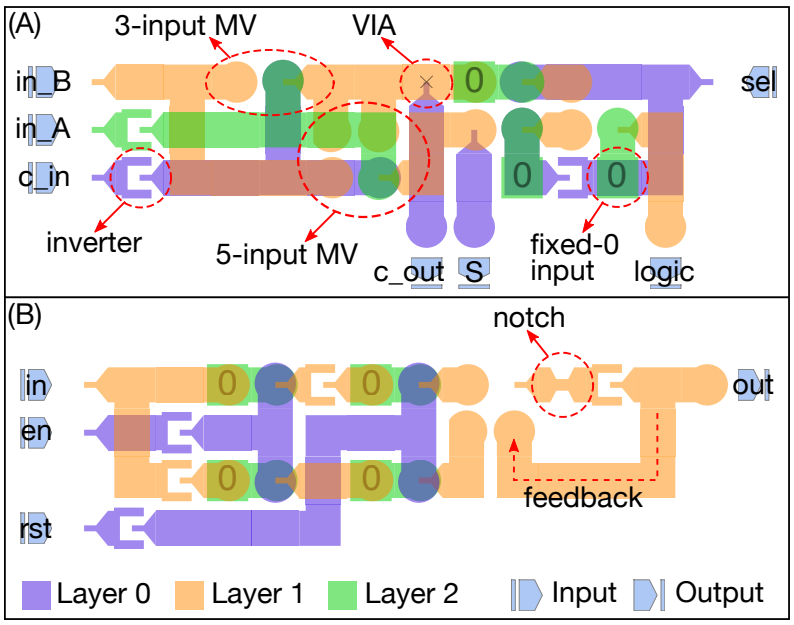

Figure 8.31: pNML CLiM cell building blocks: (A) computational element, (B) memory element.

also be programmed to execute logic operations (see subsection 8.3.2). The *sel* input controls a 2-to-1 multiplexer, whose output is *logic*, that allows to choose among *c_out* or *S* when performing logic operations. Then, one among output *S* or *logic* is chosen (by means of an additional multiplexer not shown in figure) to be sent to the memory cell. *c_out* is instead sent to the neighboring cell to create

a ripple carry adder. Some pNML basic elements are also highlighted. A 3-input majority voter (MV) is used to compute the output carry, while a 5-input MV is used to compute the sum [17]. Other highlighted elements are the inverter and the VIA that connects a nucleation center placed on another layer. Finally, fixed-0 inputs (nanomagnets with fixed magnetization) are used to fix one of the MV's inputs to obtain an AND gate. In fact, supposing that the MV has three inputs, A, B and C, it satisfies the logic function $MV = AB + BC + AC$. If $A = 0$ then $MV = BC$, which is the logic AND between inputs B and C.
The memory cell (figure 8.31(B)) exploits the notch to retain the information. The *en* signal is used to enable the writing of a new data inside the cell, whereas the *rst* signal reset the content of the cell. The output value changes accordingly to the input only when the notch is depinned. The depinning of the notch corresponds to the sampling of the input data. The feedback helps retaining the value stored in the cell when the enable signal is inactive (no writing) but the input changes.
The cell shown in figure 8.30 has two memory cells (note the presence of two notches) as the one presented in subsection 8.3.2, whose writing is controlled by signals *en* and *wr*. The *copy* signal is used to copy the content of one memory cell in the other. The *shift* signal is used to enable shift operations; the input *shift_in* comes from an adjacent cell and it allows to connect more cells in chain to perform shift operations.
Figure 8.32 shows a simplified version of the cell depicted in figure 8.30. This cell

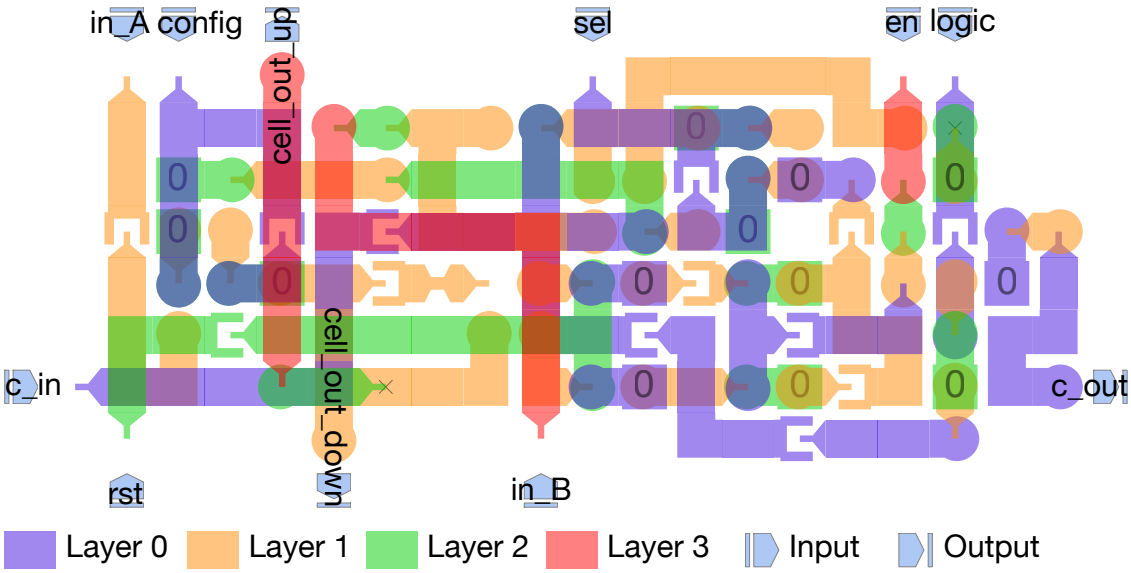

Figure 8.32: pNML CLiM cell: simplified version.

has only one memory element and it does not have signals to support shift operations.
As explained in subsection 8.3.3, when programming the CLiM array to perform

convolution, cells on even rows are used for shift operations while cells on odd rows are used for accumulation. For this reason, when designing the pNML CLiM array, the complex version of the pNML CLiM cell was used in even rows and the simplified one in odd rows in order to reduce the overall area occupied by the architecture. The pNML structure shown in figure 8.33, is a small pNML CLiM array composed of three rows. Even rows contain four complex pNML CLiM cells. The odd row contains five simple pNML CLiM cells. In fact, simplified pNML CLiM cells are more compact that complex ones and this makes it possible to fit in one row a larger number of them with respect to complex ones. This also allows to increase precision when performing accumulations.
Moreover, it can be seen that a large portion of the overall area occupied by the CLiM array is taken by interconnections. This design uses 9 layers: bottom ones are occupied by CLiM cells, whereas upper ones are used for interconnection routing. Even though the pNML CLiM array presented is very small, the layout is particularly complex because of all the connections needed between cells. This makes it very difficult to design larger structures.
The pNML CLiM array occupies an estimated area of 109.3 µm.

Table 8.5 summarized some important results obtained by considering the same methodology and assumptions presented in [11]. The values of power density and binary throughput computed for the two pNML cells presented are compared to the CMOS version of the CLiM cell (figure 8.14) when a 1-bit parallelism is considered. First of all, it can be noticed that pNML cells are far more compact than the

Cell type	**Area** [µm^2]	**Power density** [W/cm^2]	**Binary throughput** [GB/ns cm^2]	**f$_{clock}$** [MHz]
pNML simple	2.7	$3{\cdot}10^{-3}$	1.18	50
pNML complex	4.2	$1.9{\cdot}10^{-3}$	1.85	50
CMOS	40	75.7	3.7	1500

Table 8.5

CMOS one, thanks to the 3D integrability of pNML technology. Moreover, being a very low power technology, pNML allows to obtain a power density that is three orders of magnitude smaller than the one obtained in the CMOS case. For what concerns binary throughput, which is actually a density (it is evaluated per square centimeter), is slightly lower in the pNML cells with respect to the CMOS one. However, considering that the CMOS clock frequency is 1.5 GHz and the pNML one is only 50 MHz, the throughput density achievable in the pNML case is quite

remarkable. These results suggest that pNML is an ideal candidate for Logic-in-Memory.

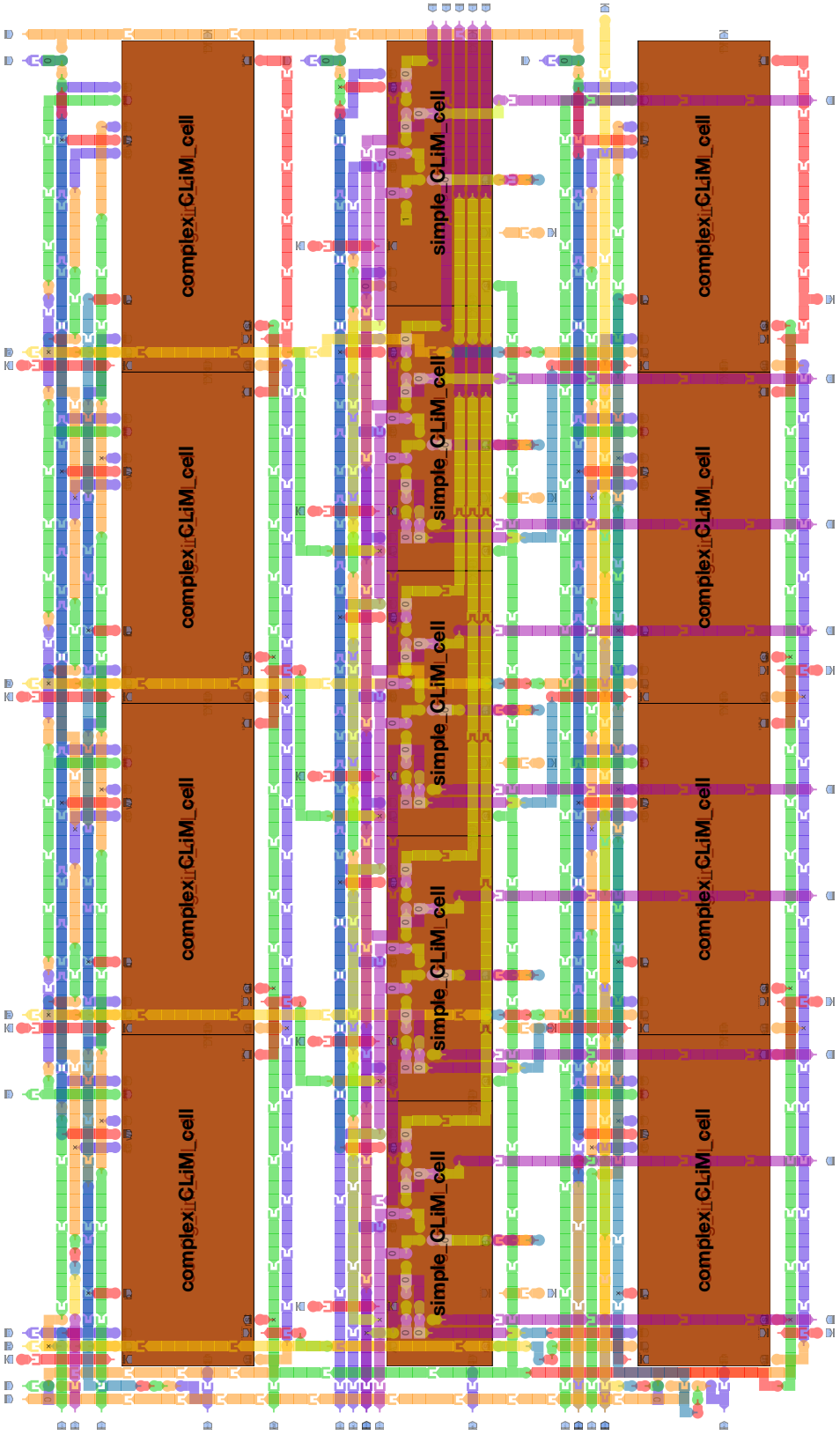

Figure 8.33: pNML CLiM array.

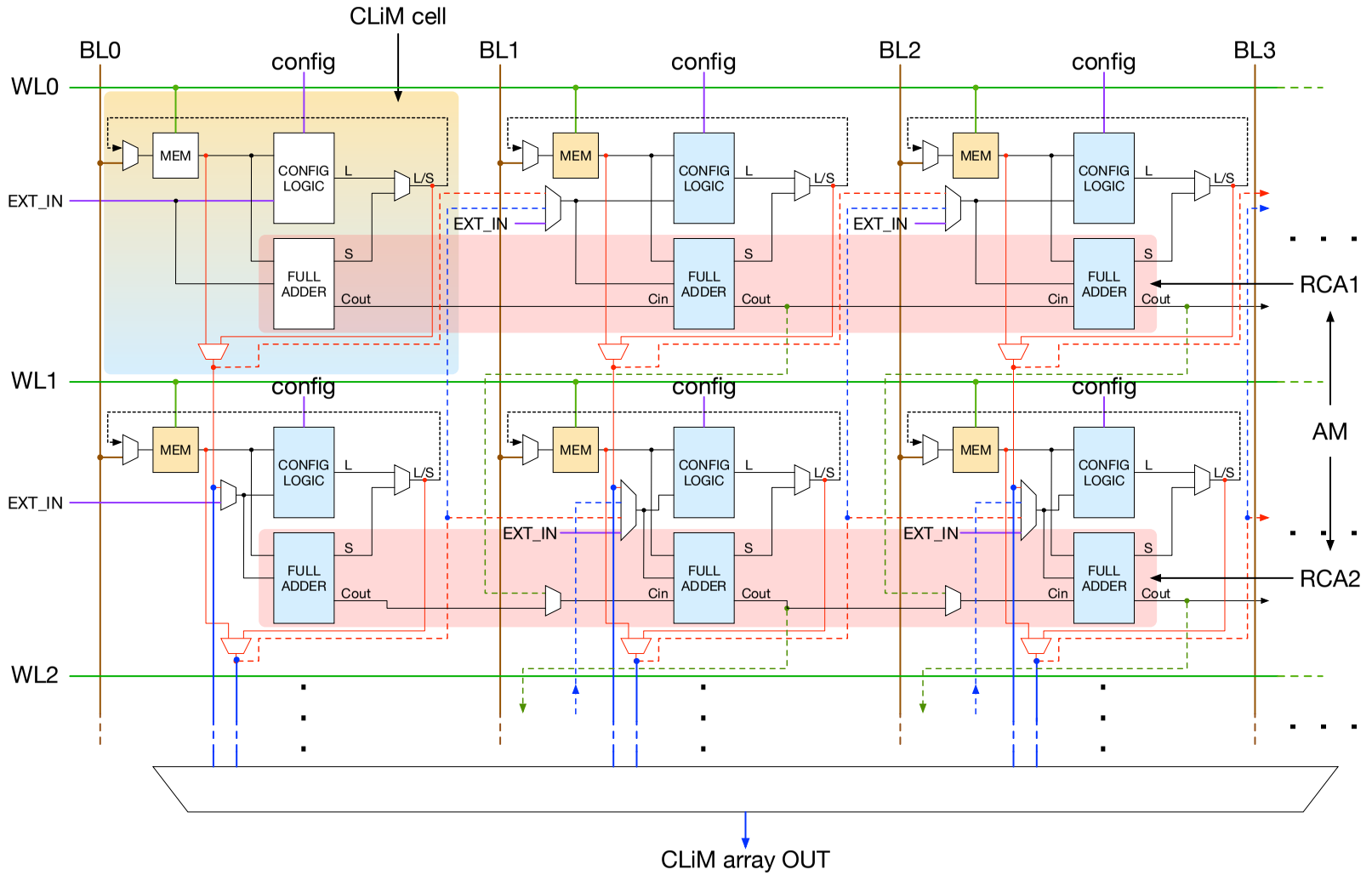

Figure 8.34: Intra-row and intra-column operations can be exploited to build complex in-memory functions. In this specific case, by exploiting intra-row operations, each memory row can work as a Ripple Carry Adder (RCA). By exploiting intra-column operations (in addition to inter-row operations), more memory rows can work as an Array Multiplier (AM).

Chapter 9

Exploration of other CLiMA Approaches

The research work conducted on CLiMA has also involved different students that with their ideas and contributions have helped defining advantages and limitations of the LiM approach. This exploratory work was carried out in the context of some thesis works and a multidisciplinary project that has involved several master's degree students. The ideas and fundamental characteristics of the CLiMA concept presented in chapter 8 have been used as basis for the study of different architectural and technological solutions. This chapter will mainly focus on the architectural considerations that can be derived from the exploration conducted. The architectures that are going to be presented in the next sections will not be described in every detail, only the main and most useful (in the context of this exploration) features will be highlighted.

9.1 CLiMA for Database Search

Database search based on bitmap indexes (subsection 8.2.1) is a perfect candidate for CLiMA as it requires the execution of simple bitwise logic operations on large quantities of data. Bitmap indexes are stored in memory in such a way that, when answering a query, operations are executed between rows. Going back to the example shown in figure 8.5, a possible mapping on CLiMA would be to store bitmap indexes on different rows, as depicted in figure 9.1. In order to perform the query 'Gender = F AND (Age = A OR Age = B)', two operations are need: a bitwise OR between the two rows containing the age range bitmap indexes and then a bitwise AND between the result of the bitwise OR and the content of the row storing the bitmap index related to the gender. From this example, some important preliminary considerations can be derived.

1. The operations required by the bitmap-based search are *logical* and *bitwise*:

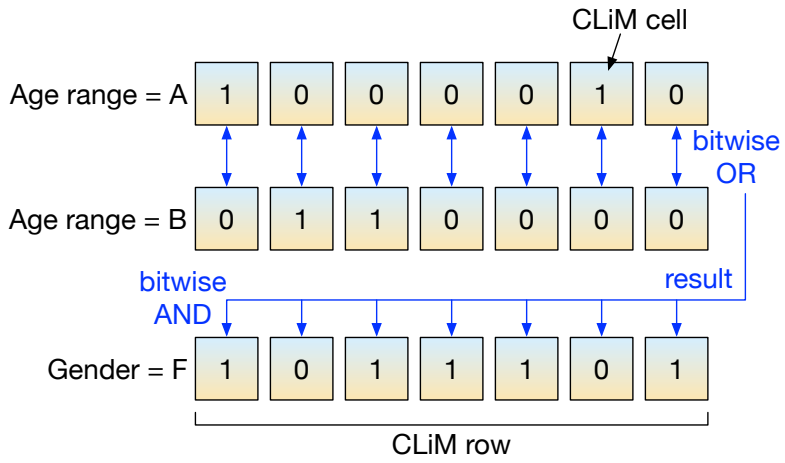

Figure 9.1: Mapping of bitmap indexes and bitwise operations on CLiMA.

this means that the building block of CLiMA, i.e. the CLiM cell, can be simply implemented as a configurable block able to perform logic operations and to store data; moreover, since operations are bitwise, cells on the same column but different rows must be directly connected.

2. Queries can be *simple* (e.g., $A + B$) or *composed* (e.g., $C \cdot (A + B)$). In the first case the operation involves only two operands (hence, two rows), while in the second case it involves more than two operands.

3. Operations can be executed on operands stored in *any position* inside the array; in some cases operands might be stored on adjacent rows, in other cases they might not.

4. If the query is composed and the operations are totally independent (e.g., $(A \cdot B) + (C \cdot D)$) then *computation can be parallelized* (e.g., $A \cdot B$ and $C \cdot D$ are executed in parallel and, at the next step, the OR between the two partial results is performed).

These considerations have driven the design choices of CLiMA for database search, whose architecture is shown in figure 9.2. The computational core of CLiMA is, as usual, the CLiM array. In this implementation, the array is a multi-bank structure whose level of parallelism depends on the number of CLiM banks in the array. Each CLiM bank, in fact, can perform computations independently from the others. Banks can also exchange data among them. In order to manage the data exchange without creating conflicts while guaranteeing flexibility, each bank is connected to a bi-directional breaker (BB block). This block routes data according to a source and a destination; as instance, if a data has to be moved from bank 0 (source) to bank 3 (destination), then BB 0 sends the data from bank 0 toward the south

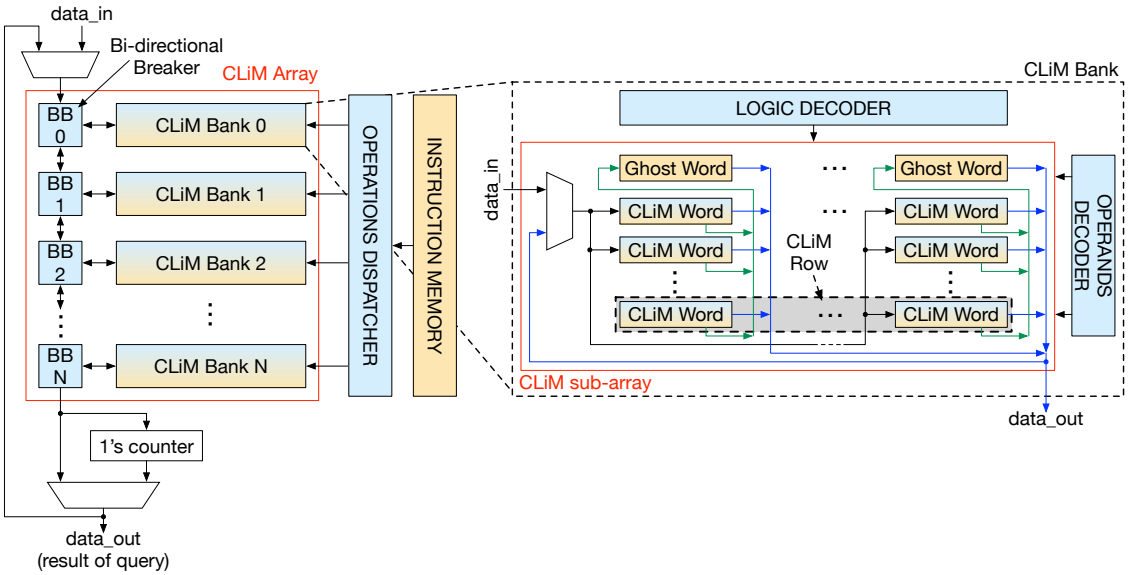

Figure 9.2: CLiMA for database search based on bitmap indexing.

direction, BBs 1 and 2 are transparent and isolate banks 1 and 2 (which are not involved in the operation) and, finally, BB 3 directs the data coming from north toward bank 3. By properly controlling bi-directional breakers data can be moved between banks with maximum flexibility.
The extra-array blocks help controlling the operations inside the CLiM array. In particular, there is an instruction memory containing instructions that are sent to the operations dispatcher block. Since CLiMA can execute more instructions in parallel, the role of the dispatcher is to assign the correct instruction to the correct bank. Each instruction specifies the type of logic operation to perform and the two operands on which performing the operation. Several execution modes are supported by the architecture:

- *single operation - single bank*: a single operation is executed between operands belonging to the same bank;
- *multi-operation - multi-bank*: multiple operations are executed in parallel either on operands belonging to the same bank or on operands belonging to different banks;
- *single composed operation*: a single composed operation is executed; composed operations involve more than two operands that can be in any bank;
- *multi-composed operation*: multiple composed operations are executed.

The one's counter is used to count the number of hits of a query of the type 'how many?', if required. The counter can be considered as *extra-row logic*.

Figure 9.2 also depicts the internal structure of a CLiM bank which is composed by a CLiM sub-array and some extra logic. The CLiM sub-array is composed of multiple words, each containing a certain number of CLiM cells. The CLiM cell has storage and logic computation capabilities. CLiM words on the same column share the same input and output lines. There are also two global input/output lines to connect the input/output column lines to the overall input/output ports of the bank. Each bank has also a ghost row composed of cells that have only storage capabilities. The ghost row is used to hold partial or final results in order to avoid overwriting the original content of the bank.
For what concerns the extra-logic, the logic decoder is used to manage the configuration of the CLiM cells according to the required logic operation. The operands decoder is instead used to activate the correct cells involved in an operation; it receives the two addresses of the operands from the operations dispatcher.
For more details on CLiMA for database search please refer to [5].

Acknowledgments I would like to thank Milena Andrighetti for the work that she has done on CLiMA for database search and for the contribution given in the context of the exploration on CLiMA.

9.2 CLiMA for Random Decision Forests

Random decision forests (subsection 8.2.2) are a good choice for in-memory implementation as they require comparisons that can be implemented with simple logic operations. The architecture of CLiMA for Random Decision Forests is shown in figure 9.3. It consists of a CLiM array composed of different sub-arrays, each storing a tree of the forest. In particular, CLiM sub-arrays store the thresholds associated to the tree's nodes, whereas the nodes memory stores information related to each node. The nodes memory is divided into multiple banks, each associated to a tree and each communicating with a CLiM sub-array. The information stored in banks are, for each node in a tree, its address and the addresses of its right and left nodes. Figure 9.4 shown the internal structure of a CLiM sub-array. The sub-array is organized in rows, each storing the threshold value associated to a node. Thresholds are unsigned values; in order to identify whether a node is a leaf or not, the MSB is set to 1 in the former case, to 0 in the latter. Each CLiM cell inside a row is composed of a logic block for performing the comparison (two logic gates) and a memory element that holds the value of the locally stored threshold. The MSB cell stores the MSB of the threshold value: if the MSB is equal to 1 then the current node is a leaf, the comparison is skipped and the result of the classification can be directly determined (blue arrow in figure 9.4 going from the MSB cell toward the class RF). If, instead the MSB 0 the current node is not a leaf and the comparison between the local threshold and the input data must be

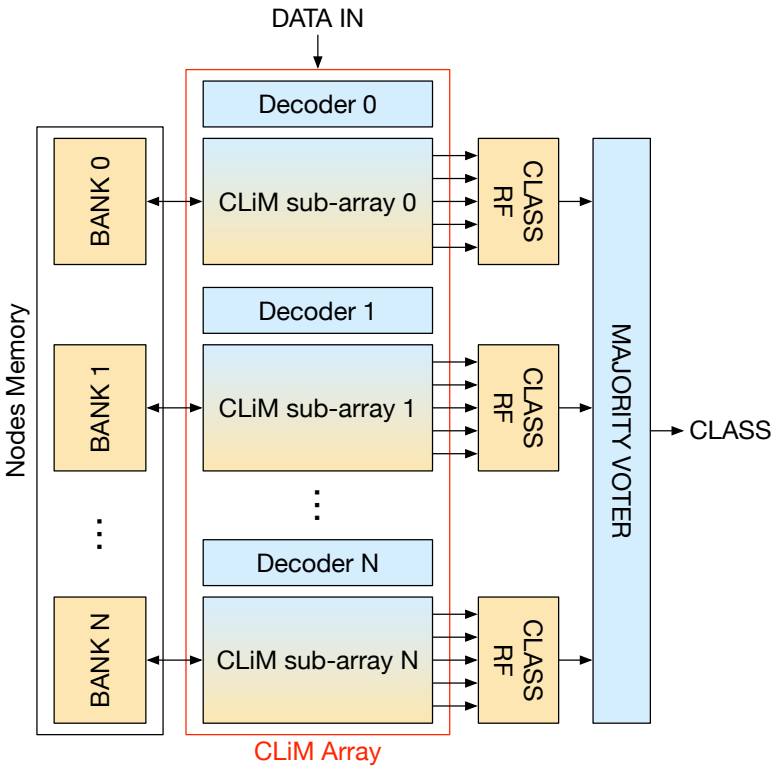

Figure 9.3: CLiMA for random decision forests.

performed. The comparison is done bit by bit inside the CLiM cells in a row. Depending on the result, the next node in the tree is chosen. In particular, if the threshold of the current node is larger than the input data than the next node is the right one, otherwise it is the left one. The result of the comparison is sent to the nodes memory and the current node is updated. In this way the whole tree is traversed until a leaf node is reached. The first leaf node reached activates the correspondent register in the class register file (RF). Each sub-array has its class RF containing the values of the classes that each tree can distinguish. Once all class values are available, the majority voter block (extra-array logic) calculates the final classification result.
For more details on CLiMA for random decision forests please refer to [116].

Acknowledgments I would like to thank Debora Uccellatore for the work that she has done on CLiMA for random decision forests and for the contribution given in the context of the exploration on CLiMA.

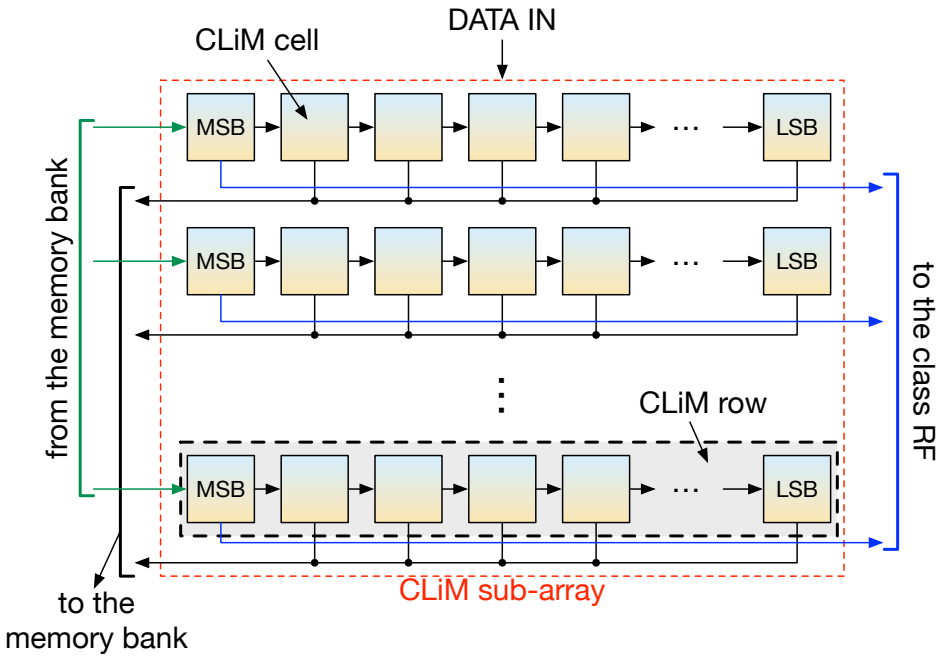

Figure 9.4: Internal structure of a sub-array.

9.3 CLiMA for AES

The AES algorithm (subsection 8.2.3) consists of different steps: some of them can be easily implemented in memory, but others cannot. This does not depend on the complexity of the operations required (all the operations are rather simple, indeed) but mainly on the complexity of the processing patterns. As instance, the *ShiftRows* step, consists of shifting the rows of the state array by a certain number of bytes to the left. This operation, per se, is not at all complex. Moreover, as shown in subsection 8.3.2, implementing in-memory shift operations is possible and effective. However, in the AES case shift operations require the movement of bytes, not single bits. For this reason, implementing an in-memory byte shift would be inefficient and complex because it would require a lot of inter-cells connections which, in turn, would cause congestion. In general, the AES algorithm requires a lot of data exchange that is not suitable to be implemented in memory. On the other side, most data transformation are based on XOR operations which can be easily integrated in memory.
The architecture of CLiMA for AES in represented in figure 9.5. It is composed of the CLiM array, some extra-logic (MixColumn Block) and other storage units. The CLiM array is an heterogeneous structure composed of three sub-arrays that contain 16 8-bit CLiM cells each. CLiM cells are composed of two storage elements (one to hold the original data, the other for partial or final results) and a XOR gate.

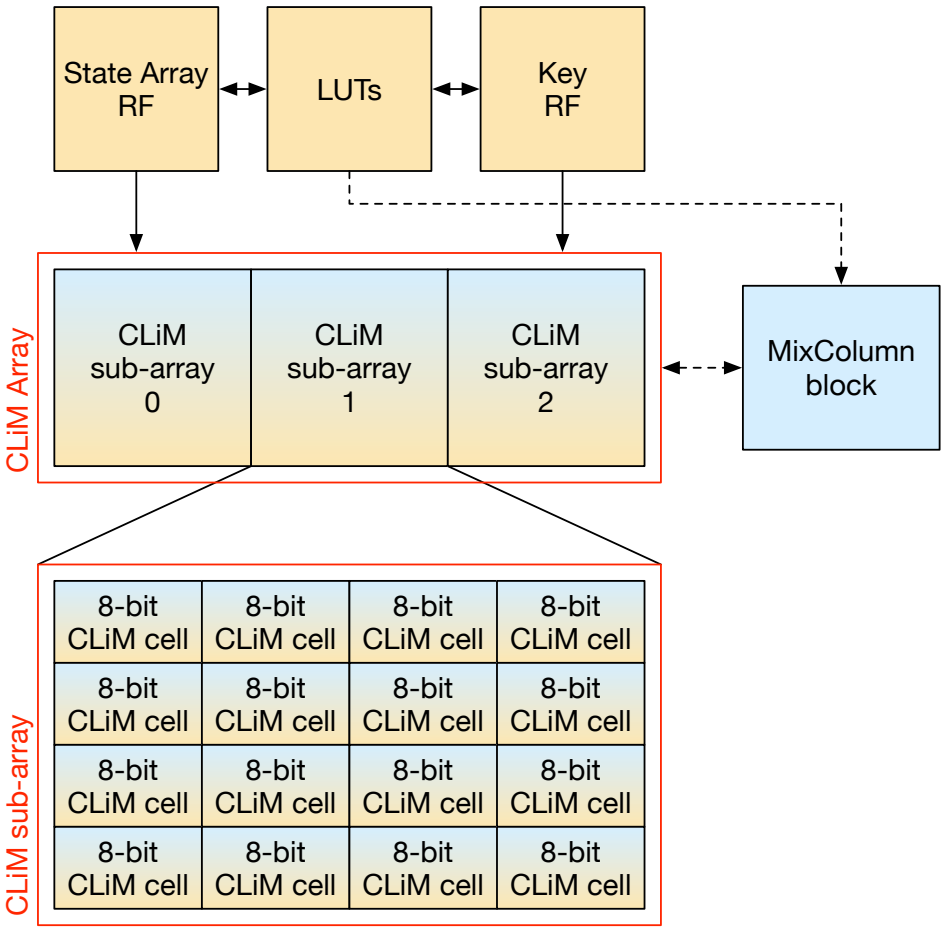

Figure 9.5: CLiMA for the Advanced Encryption Standard algorithm.

The array is heterogeneous because the sub-arrays are used for different steps of the AES. Even though all CLiM cells in sub-arrays are the same, what changes are the inter-cells connections. In particular, CLiM sub-array 0 is used perform the part of the *KeyExpansion* step that makes use of XOR operations and the *KeyAddition* step. In this case CLiM cells on different rows are connected through vertical connections because XOR operations are executed between the local stored data and the data coming from the upper cell. CLiM sub-arrays 1 and 2 instead are used for a part of the *MixColumn* step. In this case vertical inter-cell connections are not needed. Cells are connected by means of horizontal interconnections because XOR operations are executed, in this case, between the local stored data and the data coming from another cell on the same row.
Outside the CLiM array the MixColumn block is used to perform the part of the *MixColumn* step that cannot be mapped on CLiMA because it requires XOR operations between non-adjacent values. This would require a complex and congested interconnection fabric inside the array, which is unfeasible. In this case the CLiM

array is used as a memory, in fact, the MixColumn block reads data from it, executes the needed operations and writes back the updated values inside the array.
At the beginning of each round of the AES algorithm, the State Array RF (Register File) contains the data to encrypt. This data is used to access different Look-Up Tables (LUTs) when the *SubBytes* step is performed. The substituted bytes retrieved from the LUTs are written back in the State Array RF. During the write-back process bytes are also shifted in order to perform the *ShiftRows* step. For what concerns the Key RF, here the cipher key is stored. When the *KeyExpansion* step is performed, values stored in the Key RF are used to access the LUTs to perform some transformations. The transformed values are written back in the Key RF and then CLiMA is used to complete the *KeyExpansion* step, as explained earlier.
It is clear that the complexity of the AES, especially in terms of data exchanges, makes this application not perfectly suitable for a full in-memory implementation. For this reason, the architecture presented in this section uses a *mix* of LiM (the XOR operations executed inside the CLiM array), CnM (the MixColumn block that reads data from the CLiM array) and CwM (the LUTs used for byte transformations).

Acknowledgments I would like to thank Baldo Martino, Riccardo Massa and Maurizio Spada for the work done on CLiMA for AES and for the contribution given in the context of the exploration on CLiMA.

9.4 CLiMA for XNOR-Networks

Binary CNNs (subsection 8.2.4) are based on simple XOR and popcount operations that can be easily mapped on CLiMA. The architecture is depicted in figure in figure 9.6. The CLiM array is composed of different rows, each divided in a certain number of CLiM sub-rows. A detail of the internal structure of the sub-row is shown in figure 9.6 on the right. The sub-row is the basic unit of this architecture and it is composed of three simple CLiM cells. Two of them are programmed to perform XOR operations between the pixels (stored locally) and the weights (coming from the external weight memory) and one does both a XOR operation and an addition between the results coming from the other two cells. The addition performs a partial bitcount operation. The sum (S) and carry (Cout) values and then sent to the bitcount logic (extra-array logic) that calculates the complete bitcount operation. The reason why this operation is not entirely executed inside the array is that the logic needed would occupy too much space, causing the array to loose compactness. Moreover, for this implementation data unrolling was applied. The unrolling scheme is the same shown in figure 8.10, hence, weights are shared by CLiM cells placed on the same column. Accumulations are partially executed inside sub-rows in the horizontal directions by exploiting inter-cells connections.

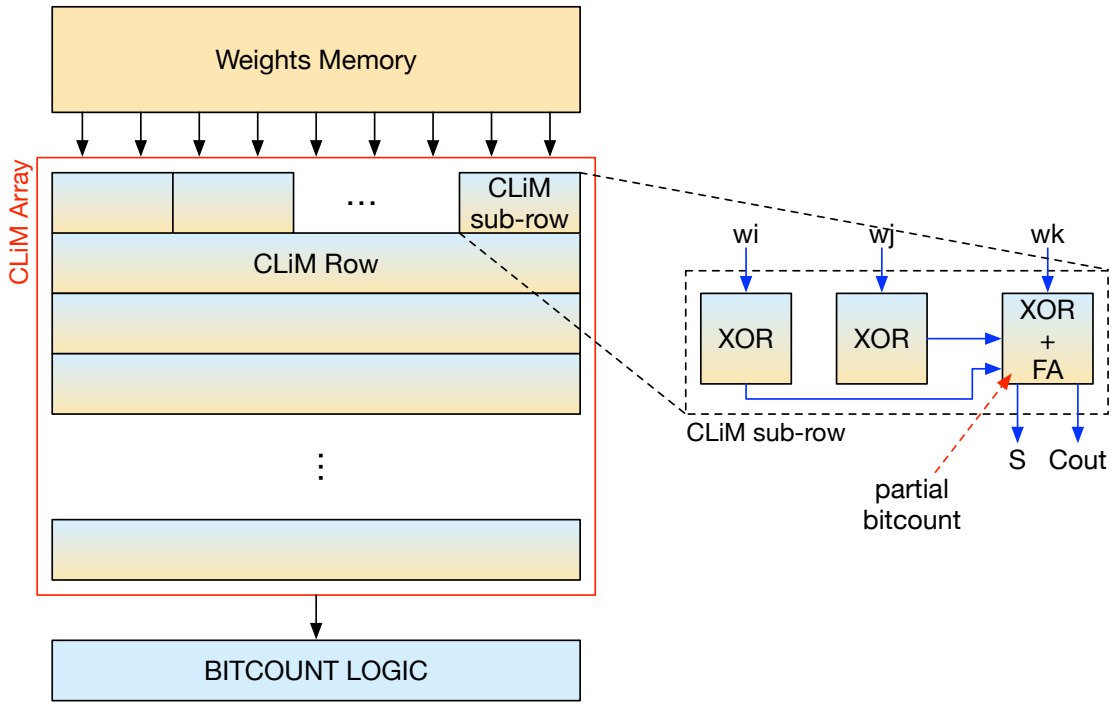

Figure 9.6: CLiMA for XNOR-Networks.

Acknowledgments I would like to thank Kristjane Koleci, Marco Marino, Luca Monterisi and Davide Tunzi for the work done on CLiMA for XNOR-Networks and for the contribution given in the context of the exploration on CLiMA.

9.5 Considerations

From the exploratory work carried out and from the description of CLiMA presented in chapter 8, different considerations can be inferred. These considerations have been of great help in establishing which features of CLiMA must be improved in order to take full advantage of the in-memory computing capabilities that the architecture can offer and in order to define the right degree of configurability that is a compromise between flexibility and efficiency.
Even though all the algorithms selected for the exploration are characterized by high-data demand, high level of parallelism and simple operations, it turns out that there are other fundamental aspects that must be taken into account.

1. *Type of operations*: it is not enough to consider the simplicity of the operations, in fact, all the algorithms selected are mainly characterized by simple operations such as logical ones, additions or shifts. However, the type of operations, even if they all require very simple hardware, has an influence on which *inter-cells connections* are needed and which are not. Moreover,

this has an influence on the *degree of in-memory computing* needed (only one among LiM, CiM, CnM, CwM or a combination of them).

2. *Data parallelism*: this factor has influence on the *granularity of the CLiM array*. As instance, if a certain algorithm requires only bitwise operations than the array will have a *fine granularity*, meaning that the basic computational cell will operate on a single bit. On the contrary, if multi-bit operations are needed (e.g. N-bit addition) then the array will be characterized by a *course granularity* and the basic computational cell will operate on multiple bits.

3. *Regularity of the data flow*: CLiMA is intrinsically characterized by a rather regular structure; its computing core is, indeed, an array which has itself a regular structure. The array favors small data movements or operations between relatively close data. Large distance data movements inside CLiMA can be supported but they are not efficient. As a result, the data flow of an algorithm has, of course, a strong effect on the interconnections inside the array.

For what concerns the type of operations, a solution to support as many operations as possible inside CliMA has already been proposed in chapter 8. Indeed, the CLiM cell presented can support logic operations, additions/subtractions and shifts. All the algorithms selected are based on a sub-set of these operations, hence, they can all be supported.
However, for what concerns data parallelism, the CLiM cell presented in subsection 8.3.2 is not dynamic, meaning that once the parallelism has been fixed it cannot be changed, therefore, operations such as additions or shifts can only be done on a fixed data parallelism. By changing the application the data parallelism changes as well, of course.
For what concerns, instead, inter-cells connections, these are what varies the most when changing the algorithm because, depending on the data flow and on how data are mapped inside the array, the interactions between cells change.
All these issues will be tackled in the next chapter and some possible solutions and future developments will be identified.

Chapter 10

Conclusions and Future Works

The research work conducted in the contest of in-memory computing has brought to the definition of CLiMA, a Configurable Logic-in-Memory Architecture that integrates in-memory computing capabilities, configurability and parallelism. Different variations of CLiMA have been presented in chapters 8 and 9 and while all are based on the same ideas and concepts described in chapter 8, a lot of work still needs to be done in order to delineate an architecture capable of adapting perfectly to the needs of different (but with similar characteristics) applications. The long-term vision is, in fact, to reach the definition of a CLiMA that is sufficiently flexible to adapt to the different requirements that each application demand. It is important to underline that the aim is not to define a general purpose version of CLiMA, but an architecture that well support all those applications that can really take advantage from a computing unit like CLiMA. In order to do so, the limitations exposed in sections 8.4 and 9.5 must be overcome.
One of the main and problematic issues is represented by interconnections inside the CLiM array. As already explained, interconnections limit the possible computation patterns, hence, the type of applications that can be run on CLiMA. A possible solution to this problem is the definition of a configurable interconnection network that can be re-configured depending on the algorithm. This has been partially achieved in CLiMA for database search (section 9.1) where bus breakers are used to flexibly route data where needed. However, this level of granularity is not enough. A more fine interconnection system is needed and a possible path might be to design a multi-level reconfigurable interconnection network with different degrees of granularity at each level in order to increase the flexibility. It is clear that the complexity of such interconnection system would be quite significant, therefore, possible computation patterns must be somehow limited.
Another limitation is represented by data parallelism. Inter-cells connection flexibility is, again, the obvious solution if the architecture has to support dynamic data parallelism. However, supporting any data parallelism is unfeasible because it would mean controlling each inter-cell connection singularly. A more reasonable

solution would be to define a parallelism granularity (e.g., multiples of 4 bits).
In section 8.4 an important issue was highlighted: the control of CLiMA is quite complex because operations and data exchanges between cells must be carefully coordinated in order to fulfill the correct computation flow. In this regard, it would be of extreme importance to build a compiler for CLiMA in order to make it easier and more straightforward the management of the computation flow inside the array.
In addition to these limitations, one of the main difficulties encountered during the extraction of the results was the impossibility of comparing CLiMA to other architectures that exploit, on some level, the concept of in-memory computing. This depends partially on the architectural differences between CLiMA and other works, but mainly on the lack of details on how the computation is carried out in other architectures and, in most cases, on the unavailability of common comparison figures. An obvious solution to this problem is to try extracting, somehow, useful common figures or data from works found in literature. However, the risk of doing so is that the information/data extracted are not precise, negatively affecting, as a consequence, the results.

All the issues and limitations that affects CLiMA open new paths for future developments. As demonstrated, the in-memory computing paradigm, as an alternative to the conventional von Neumann one, seems to be promising. This paradigm shift has an effect on both the architectural and the technological level of a computing system. This work has mainly focused on the architectural level, trying to delineate strong points and limitations of the in-memory computing paradigm in order to define a Configurable Logic-in-Memory Architecture that takes full advantage of this new paradigm to tackle the memory bottleneck problem.

Appendix A

Classification of in-Memory Computing Related Works

This appendix reports a table that collects the main and most relevant works (not all of them) found in literature related to in-memory computing and all of its shades. The works are listed in year-of-publishing order and for each of them the following items have been reported:

- Paper Ref.: bibliographical reference of the paper;
- Description: concise description of the proposed architecture;
- Technology used;
- Target applications;
- Software/models used for simulation and/or evaluation;
- Silicon proven: if the proposed architecture is silicon-proven or not;
- Approach: classification according to one of the approaches presented in chapter 7.

The name of the approaches has been shortened. The correspondent acronyms used are:

- CnM: Computation-near-Memory;
- CiM: Computation-in-Memory;
- CwM: Computation-with-Memory;
- LiM: Logic-in-Memory.

All the information found in this table have been retrieved from the referenced papers, according to what the authors have reported.

Paper Ref.	Description	Technology	Target applications	Software/models used	Silicon proven	Approach	Year
[30]	Host processor + multiple PIM (Processing-In-Memory) co-processors	180nm TSMC CMOS	Irregular data access patterns, dense matrix computations	in-house simulator based on RSIM event-driven simulator	yes	**CnM**	2002
[106]	General purpose PIM-based massively parallel architecture	DRAM + CMOS	High-Perfomance Computing (HPC)	not specified	no	**CnM**	2002
[77]	1-bit non-volatile full adder circuit mixing MTJ and CMOS	MTJ + 0.18 um CMOS	Sum	HSPICE	yes	**LiM**	2009
[124]	Multi-core architecture: custom logic layer with stacked DRAM in each core	32nm CMOS + 3D-stacked DRAM	Data intensive and sparse data: 2D-FFT, SpGEMM	in-house synthesis tool for custom LiM blocks, CACTI-3DD	no	**CnM**	2013
[123]	Host processor + PIM cores. Each core has CPU+GPU and it is connected to a HMC	22nm/16nm CMOS + HMC	Graph processing, HPC, Rodinia benchmarks	in-house ML-based model for performance and power estimation	no	**CnM**	2014
[59]	MTJ-LIM hybrid architecture. Each memory cell has a MTJ stacked above simple CMOS logic	90nm CMOS + MTJ	Database search: Hopfield NNs, Sparse clustered networks	Modelsim (VHDL), HSPICE, different Cadence tools	yes	**LiM**	2014
[84]	RRAM CAM arrays inside the FPUs of a GPU, to hold highly frequent pre-computed values	45 nm TSMC CMOS + RRAM	Image and signal processing	in-house framework to: profile apps and find high freq. patterns + generation of code to program CAMs, Multi2sim, FloPoCo, different Synopsis tool, Cadence Virtuoso	no	**CwM**	2014
[65]	8x8 array with 2 tiers: one for MIPS-like cores interconnected through a 2D-mesh network and one for 3D-stacked SRAM	130nm CMOS + 3D-stacked SRAM	AES encryption, edge detection, histogram, k-means, matrix multiplication, median filter, motion estimation, string search	Synopsis, Cadence, in-house tools for 3D-stacking	yes	**CnM**	2015
[3]	Multi-core system with 32 in-order single-issue cores with HMC on top	CMOS + HMC	Graph processing	in-house programming interface, in-house cycle-accurate x86-64 simulator	no	**CnM**	2015
[120]	Parallel SIMD RRAM-based CAM array	RRAM + 22nm CMOS	Low arithmetic intensity: N-pairs Black Scholes option pricing (BSC), N-point FFT, Dense Matrix Multiplication	in-house cycle-accurate simulator, SPICE	no	**CwM**	2015
[23]	RRAM crossbar + modified peripheral circuitry to support ANN-like computation	RRAM	ANNs	NVsim, CACTI-3DD, CACTI-IO	no	**CiM**	2016
[36]	RRAM arrays + microprogrammable control logic	RRAM	PRESENT cypher for lightweight cryptography	not specified	no	**CiM**	2016
[75]	Non-volatile memory architecture with modified circuitry to perform bitwise logic operations between memory rows	memory based on generic resistive cells	Vector-OR operations, bitmap-based graph processing and database search	HSPICE, in-house programming model to allocate data in the memory and send instructions, NVsim, CACTI-3DD, in-house simulator	no	**CiM**	2016
[61]	XNOR gates inside DRAM banks + other logic stacked below the memory using TSVs	3D-stacked DRAM + 32 nm PTM CMOS	Binary CNNs (XNOR-Net)	CACTI, Virtuoso	no	**LiM + CnM**	2017
[108]	CPU for general purpose ops + unit with 4 RRAM arrays for matrix-vector multiplications + non-volatile SRAM for data storage	150nm CMOS + RRAM	general purpose, ANNs	not specified	yes	**CiM**	2017
[43]	RRAM crossbars + logic	RRAM	graph search: BFS	NVsim	no	**CiM**	2017
[14]	3-stage pipelined GP architecture with RRAM instruction mem. + RRAM mem. for data computation and storage	RRAM	24 EPFL benchmarks	in-house algorithm to map the dataflow on the architecture and generate the instruction scheduling accordingly	no	**CiM**	2017
[64]	Multiple RRAM-based CAM arrays + microcontroller	RRAM	DNA sequence alignment	SPICE, in-house cycle-accurate simulator	no	**CwM**	2017
[100]	Modified commodity DRAM to perform bulk bitwise AND-OR-NOT operations	DRAM	Bulk bitwise logic operations (e.g. bitmap-based query processing for databases, encryption, DNA sequence mapping)	SPICE, Gem5	no	**CiM**	2017
[56]	Configurable RRAM-based architecture to realize fast in-memory adder trees	RRAM + 45nm CMOS	General OpenCL, image processing	multi2sim, Virtuoso, VTEAM memristor model	no	**CiM**	2017
[110]	HMC-based architecture where the logic layer of the HMC is used for approximate computing	Micron HMC	Big-data: Breadth-First Search, Bit Count, String Search, Bitonic Sort, K-means, K Nearest Neighbor, N Filter	multi2sim, McPAT, Cacti-3DD, HMS-Sim, in-house patch for Multi2sim to profile performance and energy	no	**CnM**	2017
[119]	Modified SRAM array in which rows of logic cells (LUT-based and XOR-based) are alternated to rows of memory cells	SRAM + 90nm SAED EDK CMOS	Data-intensive: text, hist, mask, RNG, stream, BGT, CRC, swap, string	Cadence, Synopsis DC, CACTI, Orlksim	no	**LiM**	2017
[81]	RRAM-based architecture composed of a memory crossbar that provides inputs to a heterogeneous configurable computing crossbar	RRAM	not specified	SPICE	no	**CiM**	2017
[83]	Architecture composed of different Pes (each with a resistive memory to store/compute data) + control logic	RRAM + 32 nm CMOS	Common kernels (e.g. clustering, image and signal processing)	custom mapping tool to schedule ops, in-house circuit-level macrosimulator, CACTI, Synopsis DC, Synopsis VCS	yes but only Pt/HfO2/TiN RRAM devices	**CiM**	2017

Paper Ref.	Description	Technology	Target applications	Software/models used	Silicon proven	Approach	Year
[57]	Processor composed of multiple parallel memory-based cores. Each core has different resistive CAM banks to perform LUT-based computation + small CAM that stores high freq. Patterns	RRAM	General OpenCL	in-house framework to profile applications and find high freq. patterns, Multi2sim	no	**CwM**	2017
[118]	Logic + stacked HMC	HMC + 28nm CMOS	Texture filtering	modified cycle-accurate simulator Attila, McPAT, CACTI	no	**CnM**	2017
[58]	Non-volatile query accelerator composed of different resistive crossbar memory banks for data storage/computing. The accelerator works together with a host CPU to accelerate database query processing	RRAM + 45nm CMOS	Database search on Census dataset	HSPICE, VTEAM memristor model, in-house software-based cycle-accurate simulator	no	**CwM**	2017
[52]	3D RRAM-based (different RRAM layers stacked one on top of the other) CAM units + logic for more complex operations	RRAM	DNA sequence alignment (BLAST)	NVsim, in-house c++ performance simulator	no	**CwM**	2018
[6]	MRAM-based array with modified periphery circuitry to perform simple or composed logic operations	SOT-MRAM	AES encryption	Cadence Spectre, NVSim	no	**CiM**	2018

Nomenclature

Acronyms / Abbreviations

3D-SIC 3D Stacked Integrated Circuit

AES Advanced Encryption Standard

AI Artificial Intelligence

AM Array Multiplier

ANC Artificial Nucleation Center

ANN Artificial Neural Network

ASIC Application Specific Integrated Circuit

BL Bit-Line

CAM Content Addressable Memory

CiM Computing-in-Memory

CLiMA Configurable Logic-in-Memory Architecture

CMOS Complementary Meta-Oxide Semiconductor

CnM Computing-near-Memory

CNN Convolutional Neural Network

CONV Convolution

CPU Central Processing Unit

CU Control Unit

CwM Computing-with-Memory

DBM Double Buffering Mode

DL Deep Learning

DLP Deep Learning Processor

DNN Deep Neural Network

DRAM Dynamic Random-Access Memory

EU Execution Unit

FC Fully Connected

FDSOI Fully Depleted Silicon On Insulator

FIB Focused Ion Beam

FPGA Field Programmable Gate Array

GPU Graphic Processor Unit

GUI Graphical User Interface

HBM Hybrid Bandwidth Memory

HMC Hybrid Memory Cube

IB Input Buffer

ILSVRC ImageNet Large Scale Visual Recognition Challenge

LiM Logic-in-Memory

LSB Least Significant Bit

LUT Look-Up Tables

MAC Multiply-Accumulate

MAXPOOL Max pooling

ML Machine Learning

MLP Multi-Layer Perceptron

MRAM Magnetoresistive Random-Access Memory

MSB Most Significant Bit

MTJ Magnetic Tunnel Junction

MV Majority Voter

NML Nano Magnetic Logic

NoC Network-on-Chip

OB Output Buffer

PE Processing Element

pNML Perpendicular Nano Magnetic Logic

PVT Private

RCA Ripple Carry Adder

ReLU Rectified Linear Unit

RRAM Resistive Random-Access Memory

RTL Register-Transfer Level

SA Sense Amplifier

SHD Shared

SIMD Single Instruction Multiple Data

SM Scratchpad Memory

SRAM Static Random-Access Memory

TSV Through-Silicon Via

UTBB Ultra Thin Body and BOX

VHDL VHSIC Hardware Description Language

WL Word-Line

Bibliography

[1] *2009 International Technology Roadmap for Semiconductors (ITRS)*. 2009. URL: https://www.semiconductors.org/wp-content/uploads/2018/09/Interconnect.pdf.

[2] *2013 International Technology Roadmap for Semiconductors (ITRS)*. 2013. URL: http://www.itrs2.net/2013-itrs.html.

[3] J. Ahn et al. "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing". In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. June 2015, pp. 105–117. DOI: 10.1145/2749469.2750386.

[4] Ethem Alpaydin. *Introduction to Machine Learning*. 2nd. The MIT Press, 2010. ISBN: 026201243X, 9780262012430.

[5] Milena Andrighetti. "Parallel architectures for Processing-in-Memory". Dicembre 2018. URL: http://webthesis.biblio.polito.it/9501/.

[6] S. Angizi, Z. He, and D. Fan. "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-Efficient Logic Computation". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. June 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465706.

[7] D. Apalkov, B. Dieny, and J. M. Slaughter. "Magnetoresistive Random Access Memory". In: *Proceedings of the IEEE* 104.10 (Oct. 2016), pp. 1796–1830. ISSN: 0018-9219. DOI: 10.1109/JPROC.2016.2590142.

[8] Brenna D. Argall et al. "A survey of robot learning from demonstration". In: *Robotics and Autonomous Systems* 57.5 (2009), pp. 469–483. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2008.10.024.

[9] M. Becherer et al. "A monolithic 3D integrated nanomagnetic co-processing unit". In: *Solid-State Electronics* 115 (2016). Selected papers from the EUROSOI-ULIS conference, pp. 74–80. ISSN: 0038-1101. DOI: https://doi.org/10.1016/j.sse.2015.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0038110115002324.

[10] M. Becherer et al. “Magnetic Ordering of Focused-Ion-Beam Structured Cobalt-Platinum Dots for Field-Coupled Computing”. In: *IEEE Transactions on Nanotechnology* 7.3 (May 2008), pp. 316–320. ISSN: 1536-125X. DOI: 10.1109/TNANO.2008.917796.

[11] M. Becherer et al. “Towards on-chip clocking of perpendicular Nanomagnetic Logic”. In: *Solid-State Electronics* 102 (2014). Selected papers from ESSDERC 2013, pp. 46–51. ISSN: 0038-1101. DOI: https://doi.org/10.1016/j.sse.2014.06.012. URL: http://www.sciencedirect.com/science/article/pii/S0038110114001452.

[12] Y. Bengio, A. Courville, and P. Vincent. “Representation Learning: A Review and New Perspectives”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8 (Aug. 2013), pp. 1798–1828. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2013.50.

[13] D. Bertozzi et al. “NoC synthesis flow for customized domain specific multiprocessor systems-on-chip”. In: *IEEE Transactions on Parallel and Distributed Systems* 16.2 (Feb. 2005), pp. 113–129. ISSN: 1045-9219. DOI: 10.1109/TPDS.2005.22.

[14] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay. “ReVAMP: ReRAM based VLIW architecture for in-memory computing”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. Mar. 2017, pp. 782–787. DOI: 10.23919/DATE.2017.7927095.

[15] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford University Press, Inc., 1995. ISBN: 0198538642.

[16] Y-Lan Boureau, Jean Ponce, and Yann LeCun. “A Theoretical Analysis of Feature Pooling in Visual Recognition”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, pp. 111–118. ISBN: 978-1-60558-907-7. URL: http://dl.acm.org/citation.cfm?id=3104322.3104338.

[17] Breitkreutz, Stephan et al. “1-Bit Full Adder in Perpendicular Nanomagnetic Logic using a Novel 5-Input Majority Gate”. In: *EPJ Web of Conferences* 75 (2014), p. 05001. DOI: 10.1051/epjconf/20147505001. URL: https://doi.org/10.1051/epjconf/20147505001.

[18] S. Breitkreutz et al. “Nanomagnetic Logic: Demonstration of directed signal flow for field-coupled computing devices”. In: *2011 Proceedings of the European Solid-State Device Research Conference (ESSDERC)*. Sept. 2011, pp. 323–326. DOI: 10.1109/ESSDERC.2011.6044169.

[19] L. Cavigelli and L. Benini. "Origami: A 803-GOp/s/W Convolutional Network Accelerator". In: *IEEE Transactions on Circuits and Systems for Video Technology* 27.11 (Nov. 2017), pp. 2461–2475. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2016.2592330.

[20] Yu-Hsin Chen et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks". In: *IEEE J. Solid-State Circuits* 52.1 (2017), pp. 127–138.

[21] Tianshi Chen et al. "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541967. URL: http://doi.acm.org/10.1145/2541940.2541967.

[22] Y. Chen et al. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (Jan. 2017), pp. 127–138. ISSN: 0018-9200. DOI: 10.1109/JSSC.2016.2616357.

[23] P. Chi et al. "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 27–39. DOI: 10.1109/ISCA.2016.13.

[24] M. Cofano et al. "Logic-in-Memory: A Nano Magnet Logic Implementation". In: *2015 IEEE Computer Society Annual Symposium on VLSI*. July 2015, pp. 286–291. DOI: 10.1109/ISVLSI.2015.121.

[25] Ronan Collobert et al. "Natural Language Processing (Almost) from Scratch". In: *J. Mach. Learn. Res.* 12 (Nov. 2011), pp. 2493–2537. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1953048.2078186.

[26] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3540425802.

[27] G. E. Dahl, T. N. Sainath, and G. E. Hinton. "Improving deep neural networks for LVCSR using rectified linear units and dropout". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. May 2013, pp. 8609–8613. DOI: 10.1109/ICASSP.2013.6639346.

[28] *Database Data Warehousing Guide - Oracle*. URL: https://docs.oracle.com/cd/B28359_01/server.111/b28313/indexes.htm.

[29] G. Desoli et al. "A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2017, pp. 238–239. DOI: 10.1109/ISSCC.2017.7870349.

[30] Sorin Draghici. "On the capabilities of neural networks using limited precision weights". In: *Neural Networks* 15.3 (2002), pp. 395–414. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(02)00032-1. URL: http://www.sciencedirect.com/science/article/pii/S0893608002000321.

[31] Jeff Draper et al. "The Architecture of the DIVA Processing-in-memory Chip". In: *Proceedings of the 16th International Conference on Supercomputing*. ICS '02. New York, New York, USA: ACM, 2002, pp. 14–25. ISBN: 1-58113-483-5. DOI: 10.1145/514191.514197.

[32] Morris J. Dworkin et al. *Advanced Encryption Standard (AES)*. Tech. rep. Federal Inf. Process. Stds. (NIST FIPS), 2001.

[33] I. Eichwald et al. "Towards a Signal Crossing in Double-Layer Nanomagnetic Logic". In: *IEEE Transactions on Magnetics* 49.7 (July 2013), pp. 4468–4471. ISSN: 0018-9464. DOI: 10.1109/TMAG.2013.2238898.

[34] Irina Eichwald et al. "Majority logic gate for 3D magnetic computing". In: *Nanotechnology* 25.33 (July 2014), p. 335202. DOI: 10.1088/0957-4484/25/33/335202. URL: https://doi.org/10.1088%2F0957-4484%2F25%2F33%2F335202.

[35] Irina Eichwald et al. "Signal crossing in perpendicular nanomagnetic logic". In: *Journal of Applied Physics* 115.17 (2014), 17E510. DOI: 10.1063/1.4863810. eprint: https://doi.org/10.1063/1.4863810. URL: https://doi.org/10.1063/1.4863810.

[36] A. Ferrara et al. "3D design of a pNML random access memory". In: *2017 13th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*. June 2017, pp. 5–8. DOI: 10.1109/PRIME.2017.7974093.

[37] P. Gaillardon et al. "The Programmable Logic-in-Memory (PLiM) computer". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2016, pp. 427–432.

[38] U. Garlando et al. "Architectural exploration of perpendicular Nano Magnetic Logic based circuits". In: *Integration* 63 (2018), pp. 275–282. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2018.05.001. URL: http://www.sciencedirect.com/science/article/pii/S0167926017306090.

[39] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323.

[40] M. D. Godfrey and D. F. Hendry. "The computer as von Neumann planned it". In: *IEEE Annals of the History of Computing* 15.1 (1993), pp. 11–21. ISSN: 1058-6180. DOI: 10.1109/85.194088.

[41] Jelle J. W. Goertz et al. “Domain wall depinning from notches using combined in- and out-of-plane magnetic fields”. In: *AIP Advances* 6.5 (2016), p. 056407. DOI: 10.1063/1.4944698. eprint: https://doi.org/10.1063/1.4944698. URL: https://doi.org/10.1063/1.4944698.

[42] Jiuxiang Gu et al. “Recent advances in convolutional neural networks”. In: *Pattern Recognition* 77 (2018), pp. 354–377. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.10.013.

[43] Denis A. Gudovskiy and Luca Rigazio. “ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks”. In: *CoRR* abs/1706.02393 (2017). arXiv: 1706.02393. URL: http://arxiv.org/abs/1706.02393.

[44] L. Han et al. “A novel ReRAM-based processing-in-memory architecture for graph computing”. In: *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. Aug. 2017, pp. 1–6. DOI: 10.1109/NVMSA.2017.8064464.

[45] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *CoRR* abs/1510.00149 (2015). arXiv: 1510.00149. URL: http://arxiv.org/abs/1510.00149.

[46] Song Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 243–254. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.30.

[47] Simon Haykin. *Neural Networks: A Comprehensive Foundation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2007. ISBN: 0131471392.

[48] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[49] *High Bandwidth Memory (HBM)*. URL: https://www.samsung.com/us/samsungsemiconductor/hbm/.

[50] Tin Kam Ho. “Random Decision Forests”. In: *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*. ICDAR ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 278–. ISBN: 0-8186-7128-9. URL: http://dl.acm.org/citation.cfm?id=844379.844681.

[51] J. L. Holi and J. -. Hwang. “Finite precision error analysis of neural network hardware implementations”. In: *IEEE Transactions on Computers* 42.3 (Mar. 1993), pp. 281–290. ISSN: 0018-9340. DOI: 10.1109/12.210171.

[52] Joshua Zhexue Huang. "Clustering Categorical Data with k-Modes". In: *Encyclopedia of Data Warehousing and Mining*. 2009.

[53] W. Huangfu et al. "RADAR: A 3D-ReRAM based DNA Alignment Accelerator Architecture". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. June 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465882.

[54] D. H. Hubel and T. N. Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of Physiology* 160.1 (1962), pp. 106–154. DOI: 10.1113/jphysiol.1962.sp006837.

[55] *Hybrid Memory Cube (HMC)*. URL: http://hybridmemorycube.org.

[56] S. Ikeda et al. "Magnetic Tunnel Junctions for Spintronic Memories and Beyond". In: *IEEE Transactions on Electron Devices* 54.5 (May 2007), pp. 991–1002. ISSN: 0018-9383. DOI: 10.1109/TED.2007.894617.

[57] M. Imani, S. Gupta, and T. Rosing. "Ultra-efficient processing in-memory for data intensive applications". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2017, pp. 1–6. DOI: 10.1145/3061639.3062337.

[58] M. Imani and T. Rosing. "CAP: Configurable resistive associative processor for near-data computing". In: *2017 18th International Symposium on Quality Electronic Design (ISQED)*. Mar. 2017, pp. 346–352. DOI: 10.1109/ISQED.2017.7918340.

[59] M. Imani et al. "Efficient query processing in crossbar memory". In: *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. July 2017, pp. 1–6. DOI: 10.1109/ISLPED.2017.8009204.

[60] H. Jarollahi et al. "A Nonvolatile Associative Memory-Based Context-Driven Search Engine Using 90 nm CMOS/MTJ-Hybrid Logic-in-Memory Architecture". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 4.4 (Dec. 2014), pp. 460–474. ISSN: 2156-3357. DOI: 10.1109/JETCAS.2014.2361061.

[61] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *CoRR* abs/1408.5093 (2014). arXiv: 1408.5093. URL: http://arxiv.org/abs/1408.5093.

[62] L. Jiang et al. "XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2 DRAMs". In: *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. July 2017, pp. 1–6. DOI: 10.1109/ISLPED.2017.8009163.

[63] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[64] M. Julliere. "Tunneling between ferromagnetic films". In: *Physics Letters A* 54.3 (1975), pp. 225–226. ISSN: 0375-9601. DOI: https://doi.org/10.1016/0375-9601(75)90174-7. URL: http://www.sciencedirect.com/science/article/pii/0375960175901747.

[65] R. Kaplan et al. "A Resistive CAM Processing-in-Storage Architecture for DNA Sequence Alignment". In: *IEEE Micro* 37.4 (2017), pp. 20–28. ISSN: 0272-1732. DOI: 10.1109/MM.2017.3211121.

[66] D. H. Kim et al. "Design and Analysis of 3D-MAPS (3D Massively Parallel Processor with Stacked Memory)". In: *IEEE Transactions on Computers* 64.1 (Jan. 2015), pp. 112–125. ISSN: 0018-9340. DOI: 10.1109/TC.2013.192.

[67] Judith Kimling et al. "Tuning of the nucleation field in nanowires with perpendicular magnetic anisotropy". In: *Journal of Applied Physics* 113.16 (2013), p. 163902. DOI: 10.1063/1.4802687. eprint: https://doi.org/10.1063/1.4802687. URL: https://doi.org/10.1063/1.4802687.

[68] Konstantina Kourou et al. "Machine learning applications in cancer prognosis and prediction". In: *Computational and Structural Biotechnology Journal* 13 (2015), pp. 8–17. ISSN: 2001-0370. DOI: https://doi.org/10.1016/j.csbj.2014.11.005.

[69] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[70] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541.

[71] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[72] Yann LeCun and Yoshua Bengio. "The Handbook of Brain Theory and Neural Networks". In: ed. by Michael A. Arbib. Cambridge, MA, USA: MIT Press, 1998. Chap. Convolutional Networks for Images, Speech, and Time Series, pp. 255–258. ISBN: 0-262-51102-9. URL: http://dl.acm.org/citation.cfm?id=303568.303704.

[73] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521 (May 2015), pp. 436–444. URL: https://doi.org/10.1038/nature14539.

[74] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. "Convolutional networks and applications in vision". In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems* (2010), pp. 253–256.

[75] Hui Li. *Which machine learning algorithm should I use?* Apr. 2017. URL: https://blogs.sas.com/content/subconsciousmusings/2017/04/12/%09%09machine-learning-algorithm-use/.

[76] S. Li et al. "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2016, pp. 1–6. DOI: 10.1145/2897937.2898064.

[77] P. K. Mallapragada et al. "SemiBoost: Boosting for Semi-Supervised Learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.11 (Nov. 2009), pp. 2000–2014. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2008.235.

[78] S. Matsunaga et al. "MTJ-based nonvolatile logic-in-memory circuit, future prospects and issues". In: *2009 Design, Automation Test in Europe Conference Exhibition*. Apr. 2009, pp. 433–435. DOI: 10.1109/DATE.2009.5090704.

[79] Shoun Matsunaga et al. "Fabrication of a Nonvolatile Full Adder Based on Logic-in-Memory Architecture Using Magnetic Tunnel Junctions". In: *Applied Physics Express* 1 (Aug. 2008), p. 091301. DOI: 10.1143/apex.1.091301. URL: https://doi.org/10.1143%2Fapex.1.091301.

[80] B. Moons et al. "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2017, pp. 246–247.

[81] D. E. Nikonov and I. A. Young. "Benchmarking of Beyond-CMOS Exploratory Devices for Logic Integrated Circuits". In: *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 1 (Dec. 2015), pp. 3–11. ISSN: 2329-9231. DOI: 10.1109/JXCDC.2015.2418033.

[82] G. Papandroulidakis et al. "Crossbar-Based Memristive Logic-in-Memory Architecture". In: *IEEE Transactions on Nanotechnology* 16.3 (May 2017), pp. 491–501. ISSN: 1536-125X. DOI: 10.1109/TNANO.2017.2691713.

[83] V. Peluso et al. "Ultra-Fine Grain Vdd-Hopping for energy-efficient Multi-Processor SoCs". In: *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. Sept. 2016, pp. 1–6. DOI: 10.1109/VLSI-SoC.2016.7753580.

[84] W. Qian et al. “Energy-Efficient Adaptive Computing With Multifunctional Memory”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 64.2 (Feb. 2017), pp. 191–195. ISSN: 1549-7747. DOI: 10.1109/TCSII.2016.2554958.

[85] A. Rahimi et al. “Energy-efficient GPGPU architectures via collaborative compilation and memristive memory-based computing”. In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. June 2014, pp. 1–6. DOI: 10.1109/DAC.2014.6881522.

[86] Mohammad Rastegari et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 525–542. ISBN: 978-3-319-46493-0.

[87] Mohammad Rastegari et al. “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks”. In: *CoRR* abs/1603.05279 (2016). arXiv: 1603.05279. URL: http://arxiv.org/abs/1603.05279.

[88] F. Riente et al. “MagCAD: A Tool for the Design of 3D Magnetic Circuits”. In: *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* 3 (2017), pp. 65–73. DOI: 10.1109/JXCDC.2017.2756981.

[89] F. Riente et al. “Towards Logic-In-Memory circuits using 3D-integrated Nanomagnetic logic”. In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. Oct. 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738700.

[90] Fabrizio Riente et al. “Controlled data storage for non-volatile memory cells embedded in nano magnetic logic”. In: *AIP Advances* 7.5 (2017), p. 055910. DOI: 10.1063/1.4973801. eprint: https://doi.org/10.1063/1.4973801. URL: https://doi.org/10.1063/1.4973801.

[91] Maximilian Riesenhuber and Tomaso Poggio. “Hierarchical models of object recognition in cortex”. In: *Nature Neuroscience* 2 (Nov. 1999), pp. 1019–1025. URL: https://doi.org/10.1038/14819.

[92] Frank Rosenblatt. *The perceptron – A perceiving and recognizing automaton*. Tech. rep. Cornell Areonautical Laboratory, INC., 1957.

[93] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *CoRR* abs/1409.0575 (2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[94] “Supervised Learning”. In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 941–941. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_803. URL: https://doi.org/10.1007/978-0-387-30164-8_803.

[95] "Unsupervised Learning". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 1009–1009. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_867. URL: https://doi.org/10.1007/978-0-387-30164-8_867.

[96] A. L. Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of Research and Development* 44.1.2 (Jan. 2000), pp. 206–226. ISSN: 0018-8646. DOI: 10.1147/rd.441.0206.

[97] G. Santoro et al. "Design-Space Exploration of Pareto-Optimal Architectures for Deep Learning with DVFS". In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351685.

[98] G. Santoro et al. "Energy-performance design exploration of a low-power microprogrammed deep-learning accelerator". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2018, pp. 1151–1154. DOI: 10.23919/DATE.2018.8342185.

[99] G. Santoro et al. "Exploration of multilayer field-coupled nanomagnetic circuits". In: *Microelectronics Journal* 79 (2018), pp. 46–56. ISSN: 0026-2692. DOI: https://doi.org/10.1016/j.mejo.2018.06.014. URL: http://www.sciencedirect.com/science/article/pii/S0026269217309527.

[100] R. Sarikaya, G. E. Hinton, and A. Deoras. "Application of Deep Belief Networks for Natural Language Understanding". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 22.4 (Apr. 2014), pp. 778–784. ISSN: 2329-9290. DOI: 10.1109/TASLP.2014.2303296.

[101] Vivek Seshadri et al. "Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology". In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 '17. Cambridge, Massachusetts: ACM, 2017, pp. 273–287. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3124544. URL: http://doi.acm.org/10.1145/3123939.3124544.

[102] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. DOI: 10.1017/CBO9781107298019.

[103] D. Shin et al. "DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. Feb. 2017, pp. 240–241. DOI: 10.1109/ISSCC.2017.7870350.

[104] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–359. DOI: https://doi.org/10.1038/nature24270.

[105] J. Sim et al. "14.6 A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems". In: *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. Jan. 2016, pp. 264–265. DOI: 10.1109/ISSCC.2016.7418008.

[106] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[107] T. L. Sterling and H. P. Zima. "Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing". In: *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Nov. 2002, pp. 48–48. DOI: 10.1109/SC.2002.10061.

[108] Peter Stone. "Reinforcement Learning". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 849–851. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_714. URL: https://doi.org/10.1007/978-0-387-30164-8_714.

[109] F. Su et al. "A 462GOPs/J RRAM-based nonvolatile intelligent processor for energy harvesting IoE system featuring nonvolatile logics and processing-in-memory". In: *2017 Symposium on VLSI Technology*. June 2017, T260–T261. DOI: 10.23919/VLSIT.2017.7998149.

[110] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[111] Y. Tang et al. "ApproxPIM: Exploiting realistic 3D-stacked DRAM for energy-efficient processing in-memory". In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. Jan. 2017, pp. 396–401. DOI: 10.1109/ASPDAC.2017.7858355.

[112] Fengbin Tu. *Neural Networks on Silicon*. URL: https://github.com/fengbintu/Neural-Networks-on-Silicon.

[113] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: *Mind* LIX.236 (1950), pp. 433–460. DOI: 10.1093/mind/LIX.236.433.

[114] A. M. Turing and M. Woodger. *A. M. Turing's ACE Report of 1946 and Other Papers*. Vol. 10. Charles Babbage Institute Reprint Series. The MIT Press, Apr. 1986.

[115] G. Turvani et al. "A pNML Compact Model Enabling the Exploration of Three-Dimensional Architectures". In: *IEEE Transactions on Nanotechnology* 16.3 (May 2017), pp. 431–438. ISSN: 1536-125X. DOI: 10.1109/TNANO.2017.2657822.

[116] Debora Uccellatore. "Logic-in-Memory implementation of Random Forest Algorithm". Dicembre 2018. URL: http://webthesis.biblio.polito.it/9483/.

[117] Marian Verhelst. *Deep Learning Processor Survey*. 2017. URL: https://www.esat.kuleuven.be/micas/index.php/deep-learning-processor-survey.

[118] J. von Neumann. "First draft of a report on the EDVAC". In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. ISSN: 1058-6180. DOI: 10.1109/85.238389.

[119] C. Xie et al. "Processing-in-Memory Enabled Graphics Processors for 3D Rendering". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 637–648. DOI: 10.1109/HPCA.2017.37.

[120] K. Yang, R. Karam, and S. Bhunia. "Interleaved logic-in-memory architecture for energy-efficient fine-grained data processing". In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. Aug. 2017, pp. 409–412. DOI: 10.1109/MWSCAS.2017.8052947.

[121] L. Yavits et al. "Resistive Associative Processor". In: *IEEE Computer Architecture Letters* 14.2 (July 2015), pp. 148–151. ISSN: 1556-6056. DOI: 10.1109/LCA.2014.2374597.

[122] S. Yin et al. "A 1.06-to-5.09 TOPS/W reconfigurable hybrid-neural-network processor for deep learning applications". In: *2017 Symposium on VLSI Circuits*. June 2017, pp. C26–C27. DOI: 10.23919/VLSIC.2017.8008534.

[123] S. Yu. *Resistive Random Access Memory (RRAM)*. Morgan & Claypool, 2016. ISBN: 9781627059305. URL: https://ieeexplore-ieee-org.ezproxy.biblio.polito.it/document/7437543.

[124] Dongping Zhang et al. "TOP-PIM: Throughput-oriented Programmable Processing in Memory". In: *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: ACM, 2014, pp. 85–98. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600213.

[125] Q. Zhu et al. "A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing". In: *2013 IEEE International 3D Systems Integration Conference (3DIC)*. Oct. 2013, pp. 1–7. DOI: 10.1109/3DIC.2013.6702348.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.